KTH Industrial Engineering
and Management

Specifying Safety-Critical Heterogeneous Systems Using Contracts Theory

JONAS WESTMAN

Doctoral Thesis
Stockholm, Sweden 2016

TRITA MMK 2016-05
ISSN 1400-1179
ISRN KTH/MMK/R–16/05–SE
ISBN 978-91-7729-106-0

KTH School of Industrial
Technology and Management
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i maskinkonstruktion torsdagen den 29 september 2016 klockan 09.00 i Kollegiesalen, Kungl Tekniska högskolan, Brinellvägen 8, Stockholm.

Tryck: Universitetsservice US AB

Abstract

Requirements engineering (RE) is a well-established practice that is also emphasized in safety standards such as IEC 61508 and ISO 26262. Safety standards advocate a particularly stringent RE where requirements must be structured in an hierarchical manner in accordance with the system architecture; at each level, requirements must be allocated to heterogeneous (SW, HW, mechanical, electrical, etc.) architecture elements and trace links must be established between requirements. In contrast to the stringent RE in safety standards, according to previous studies, RE in industry is in general of poor quality. Considering a typical RE tool, other than basic impact analysis, the tool neither gives feedback nor guides a user when specifying, allocating, and structuring requirements. In practice, for industry to comply with the stringent RE in safety standards, better support for RE is needed, not only from tools, but also from principles and methods.

Therefore, a foundation is presented consisting of an underlying theory for specifying heterogeneous systems and complementary principles and methods to specifically support the stringent RE in safety standards. This foundation is indeed suitable as a base for implementing guidance- and feedback-driven tool support for such stringent RE; however, the fact is that the proposed theory, principles, and methods provide essential support regardless if tools are used or not.

The underlying theory is a formal compositional contracts theory for heterogeneous systems. This contracts theory embodies the essential RE property of separating requirements on a system from assumptions on its environment. Moreover, the contracts theory formalizes the stringent RE effort of structuring requirements hierarchically with respect to the system architecture. Thus, the proposed principles and methods for supporting the stringent RE in safety standards are well-rooted in formal concepts and conditions, and are thus, theoretically sound. Not only that, but the foundation is indeed also tailored to be enforced by both existing and new tools considering that the support is based on precise mathematical expressions that can be interpreted unambiguously by machines. Enforcing the foundation in a tool entails support that guides and gives feedback when specifying heterogeneous systems in general, and safety-critical ones in particular.

Keywords: Contracts, Heterogeneous Systems, Safety, Architecture, Requirements, Specification, Elements, Compositional, IEC 61508, ISO 26262.

Sammanfattning

Kravhantering är en väletablerad praxis som också betonas i säkerhetsstandarder såsom IEC 61508 och ISO 26262. Säkerhetsstandarder förespråkar en särskilt noggrann kravhantering där krav skall struktureras på ett hierarkiskt sätt i enlighet med systemarkitekturen; på varje nivå så skall krav allokeras till heterogena (SW, HW, mekaniska, elektriska, etc.) arkitekturelement och spårlänkar skall upprättas mellan kraven. I motsats till den noggranna kravhanteringen i säkerhetsstandarder så är kravhantering i industrin av allmänt dålig kvalitet enligt tidigare studier. Ett typisk kravverktyg ger inte mycket mer stöd än grundläggande konsekvensanalyser, d.v.s. verktyget ger varken återkoppling eller vägledning för att formulera, allokera, och strukturera krav. Bättre stöd behövs för att industrin i praktiken skall kunna förverkliga den noggranna kravhanteringen i säkerhetsstandarder – inte bara stöd från verktyg, men också från kravhanteringsprinciper och metoder.

Därför presenteras ett fundament bestående av en underliggande teori för specifiering av heterogena system, samt kompletterande principer och metoder för att stödja den noggranna kravhanteringen i säkerhetsstandarder. Detta fundament är lämplig som en bas för att kunna implementera verktyg som ger återkoppling och vägledning för kravhantering; likväl så ger den föreslagna teorin, principerna och metoderna essentiellt stöd oavsett om verktyg används eller inte.

Den underliggande teorin är en kompositionell och formell kontraktsteori för heterogena system. Denna kontraktsteori ger konkret form åt den centrala kravhanteringsegenskapen att separera kraven på ett system från antaganden på dess omgivning. Dessutom så formaliserar kontraksteorin den noggranna uppgiften att hierarkiskt strukturera krav i enlighet med systemarkitekturen. Således så är de föreslagna principerna och metoderna för att stödja den noggranna kravhanteringen i säkerhetsstandarder välförankrade i formella begrepp och villkor och är därmed också teoretiskt sunda. Det erbjudna stödet är dessutom välanpassat för att kunna verkställas av såväl befintliga som nya verktyg med tanke på att stödet är grundat på exakta matematiska uttryck som kan tolkas entydigt av maskiner. Verkställandet av fundamentet av ett verktyg medför stöd i form av vägledning och återkoppling vid specifiering av heterogena system i allmänhet, och säkerhetskritiska sådana i synnerhet.

Nyckelord: Kontrakt, Heterogena System, Säkerhet, Arkitektur, Kravhantering, Specifiering, Element, Kompositionell, IEC 61508, ISO 26262.

Acknowledgements

First of all, I would like to give my sincere thanks to my main supervisor Mattias Nyberg. I can honestly not think of a person, real or conceptual, who could have been more suitable as a supervisor for me than him.

The work has been performed as part of the collaborative industrial research project ESPRESSO between Scania CV AB in Södertälje and the division of Mechatronics, Department of Machine Design, KTH.

From KTH, I would like to thank my assistant supervisor Martin Törngren for his passion and his wonderful ability to inspire. I would also like to thank everyone at the department, and the Mechatronics division in particular, for providing an excellent working environment. Outside of the department, I thank Dilian Gurov for the fruitful discussions.

Many thanks also goes to all my colleagues at Scania for giving me the opportunity to work with real industrial challenges. I would especially like to thank Jon Andersson for giving his full support for my and Mattias' 'radical' research ideas from the very start. I also extend my gratitude to developers such as Anton Einarsson and Oscar Thydén who enabled the realization of these radical ideas in practice. This gratitude also goes to all the master thesis students that I have supervised throughout the years (and many thanks for the unique gifts I have received).

This work has been financed by Scania CV AB and received funding under grant agreement no.2011 − 04446 from *Fordonsstrategisk Forskning och Innovation* (FFI) - a partnership between the Swedish government and automotive industry for joint funding of research, innovation and development concentrating on Climate & Environment and Safety.

Finally, I would like to thank my family, especially my parents Bo and Cristina and my sister Linda, and also all my friends. You are what is important to me.

Dr.Jonas (I did it all for the title)

Contents

Lists of Papers

The following Papers A-D are included in this thesis:

A **"Conditions of Contracts for Separating Responsibilities in Heterogeneous Systems"**, Jonas Westman and Mattias Nyberg, revised version of paper submitted in February 2016 to *Formal Methods of System Design*.

Jonas was the main contributor and wrote the paper. Mattias supervised the process, reviewed the paper, and gave feedback/ideas. Other than a few reformulations of sentences with the intent to further clarify concepts, the main difference between the version in this thesis and the submitted one is in Section 4. This section was modified due to a technical inconsistency; this inconsistency was discovered by the author of this thesis himself and is resolved in the presented version in this thesis and will also not appear in the future journal version. The presented material in Paper A is an extension of the following peer-reviewed publication:

i) **"Environment-Centric Contracts for Design of Cyber-Physical Systems"**, Jonas Westman and Mattias Nyberg, *MODELS – ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems*, Valencia (2014).

The principal concepts in Paper A have also been applied in a major industrial case-study. This case-study is described in the following peer-reviewed publications:

ii) **"Contracts for Structuring and Specifying Requirements on Cyber-Physical Systems"**, Jonas Westman and Mattias Nyberg, book chapter in *Cyber-Physical Systems: From Theory to Practice*, CRC Press, 2015; and

iii) **"A Reference Example on the Specification of Safety Requirements using ISO 26262"**, Jonas Westman and Mattias Nyberg, *SAFECOMP 2013 – Workshop DECS (ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems) of the 32nd International Conference on Computer Safety, Reliability and Security*, France (2013).

B **"Extending Contract Theory with Safety Integrity Levels"**, Jonas Westman and Mattias Nyberg, *HASE - 16th IEEE International Symposium on High Assurance Systems Engineering*, Daytona Beach (2015).

Jonas was the main contributor and wrote the paper. Mattias supervised the process, reviewed the paper, and gave feedback/ideas. Basic concepts in Paper B were also applied in the industrial case study described in papers (ii) and (iii). To illustrate the application of concepts in Paper A and B in practice, the description of the case study in paper (ii) is added as an appendix to Paper B in this thesis. Paper B is based on the following peer-reviewed publication:

iv) **"Structuring Safety Requirements Using Contract theory"**,
Jonas Westman, Mattias Nyberg, and Martin Törngren, *SAFECOMP - 32nd International Conference on Computer Safety, Reliability and Security*, France (2013)

C **"Formal Architecture Modeling of Sequential Non-Recursive C Programs"**, Jonas Westman, Mattias Nyberg, Joakim Gustavsson, and Dilian Gurov, submitted in April 2016 to *Science of Computer Programming*.

Jonas was the main contributor and wrote all the sections of the paper except for Section 5, which was written by Joakim. Under the supervision of Jonas, Joakim also developed the tool implementation. Mattias reviewed and gave feedback/ideas. Dilian reviewed Section 2 and gave feedback. Paper C was submitted to *Science of Computer Programming* following an invitation to write an extension of the following peer-reviewed publication:

v) **"Formal Architecture Modeling of Sequential C-Programs"**,
Jonas Westman and Mattias Nyberg, *FACS - 12th International Conference on Formal Aspects of Component Software*, Rio De Janeiro (2015)

D **"Providing Tool Support for Specifying Safety-Critical Systems by Enforcing Syntactic Contract Conditions"**, Jonas West-

man and Mattias Nyberg, extended version of paper submitted in September 2016 to *Requirements Engineering*.

Jonas was the main contributor and wrote the paper. Mattias reviewed the paper and gave feedback/ideas. Paper D is an extension and generalization of the following peer-reviewed paper:

vi) **"CPS Specifier – A Specification Tool for Safety-Critical Cyber-Physical Systems"**, Jonas Westman and Mattias Nyberg, accepted for publication in *CyPhy 2016 Sixth International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems*, Pittsburgh (2016)

Other than appended Papers A-D and the related papers (i)-(vi), other peer-reviewed publications where the author of this thesis is coauthor, are:

- **"Failure Propagation Modeling based on Contracts Theory"**, Mattias Nyberg and Jonas Westman, *EDCC - 11th European Dependable Computing Conference*, France (2015)
- **"Experience on applying software architecture recovery to automotive embedded systems"**, Xinhai Zhang, Magnus Persson, Mattias Nyberg, Bahrooz Mokhtari, Anton Einarson, Hans Linder, Jonas Westman, DeJiu Chen, and Martin Törngren, *CSMR-WCRE – IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering* (2014)
- **"A characterization of integrated multi-view modeling in the context of embedded and cyber-physical systems"**, Magnus Persson, Martin Törngren, Ahsan Qamar, Jonas Westman, Matthias Biehl, Stavros Tripakis, Hans Vangheluwe, and Joachim Denil, *EMSOFT - International Conference on Embedded Software* (2013)

Specifying Safety-Critical Heterogeneous Systems Using Contracts Theory

1 Introduction

Requirements engineering (RE) [1, 2] 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 [3]. In fact, the general FuSa standard IEC 61508 [4] 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) [3–6] that specifies the required *reliability* [7] of a system or component, in order to achieve a tolerable level of risk. FuSa is a key property of *heterogeneous systems* [8, 9], i.e. systems such as airplanes and automotive vehicles that are composed of parts from different domains, specifically software (SW), hardware (HW), and physical [10] (mechanical, electrical, etc.).

In IEC 61508 and its derivative FuSa standards such as ISO 26262 [3] for the automotive domain, safety requirements must be structured in an hierarchical manner in accordance with the *system architecture* [11]; at each level, safety requirements must be *allocated* to *architecture elements* and *trace links* [12] must be established between requirements on different levels. According to ISO 26262, this structuring must be done from an overall system level all the way down to requirements on atomic SW and HW elements. 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"* [3]. Establishing completeness means that it can be inferred that overall system-level requirements are satisfied, from verifying that atomic elements satisfy their requirements; such an indirect verification approach is called *compositional verification* [13, 14].

Note that compositional verification is typically associated with *formal* verification techniques. In this thesis, the term *compositional verification* will instead be used to characterize the overall verification approach, regardless of whether the verification techniques used are formal or not.

In general, compositional verification allows verification of complex systems when it is infeasible, due to e.g. required effort/cost, to use a direct verification approach to ensure that overall system-level requirements are satisfied. Not only that, but compositional verification also has other benefits. For example, in [15, 16], compositional verification is proposed as a means of verifying software product lines to manage the explosion of variants

in highly variable systems. Additionally, the work in [17, 18] shows how to model failure propagation by considering the manner in which requirements are structured for compositional verification.

2 Challenges and Current Situation in Industry

Section 1 described RE in the context of FuSa standards, in particular how safety requirements are structured for compositional verification. Additionally, many potential benefits from using compositional verification was presented.

While offering many potential benefits, compositional verification does indeed require the effort of establishing lower-level requirements that can be satisfied by elements of a system and that are complete with respect to higher-level requirements. To achieve this, it is necessary that requirements are of high quality, considering attributes such as readability, unambiguity, verifiability, etc. In turn, this requires a high degree of *stringency* when specifying requirements; however, specifying high quality requirements is not sufficient for establishing completeness, which also requires a high degree of stringency when allocating requirements and establishing trace links between them.

In contrast to the stringent RE required for compositional verification, requirements in industry are in general of poor quality [19] and are typically incomplete [20], and this is also true for safety requirements [20, 21]. Considering a typical RE tool such as IBM Rational DOORS, which is extensively used in industry [1], other than basic impact analysis, the tool neither gives feedback nor guides a user when specifying, allocating, and structuring requirements; hence, a property such as completeness must be established without any concrete support from the tool.

That is, in a typical industrial development environment as exemplified above, increasing the level of stringency of RE to the degree where it is possible to verify a property such as completeness, means increasing only the manual effort, i.e. man-hours, spent on RE. Thus, considering the previously mentioned benefits from using compositional verification, in practice, these benefits must be weighed against such a manual effort. In the end, increasing the manual effort spent on RE might not be justifiable; that is, it is possible that a better product is produced, considering stakeholder concerns such as cost, performance, and even FuSa, if this effort is directed elsewhere [22].

However, regardless if it is justifiable or not to spend additional manual effort on RE, stringent RE is indeed required in order to comply with FuSa standards; in fact, ISO 26262 even requires that completeness is to be verified. Thus, these standards push heterogeneous systems developers into putting more effort on RE and, without better support, this effort will constitute solely of tasks performed manually by engineers.

3 Addressing Challenges Using Contracts Theory

Consider the gap described in Section 2 between RE in industry and the stringent RE needed to comply with FuSa standards. The view in [23], which is shared in this thesis, is that RE is a complex and error-prone process that can benefit from more intelligent support. In fact, in order for industry to be able to comply with the stringent RE in safety standards in practice, better support is crucial; not just support in tools, but also in the form of principles and methods for RE.

Therefore, this thesis presents a foundation, consisting of an underlying theory for specifying heterogeneous systems, complemented with principles and methods that describe how to apply the theory in order to support the stringent RE advocated in FuSa standards. As will be shown in this thesis, this foundation is indeed suitable as a base for implementing tool support that reduces the manual effort needed to fill the gap described in Section 1 by automating tasks and providing guidance and feedback when specifying a heterogeneous systems. However, regardless if tools are used or not, the proposed theory, principles, and methods, provide essential support when specifying heterogeneous systems in general, and safety-critical ones in particular.

3.1 Starting Point in Literature

Prior to describing the novelties of this foundation in Section 3.2, this section describes and motivates the considered starting point of the proposed foundation in existing literature. In order to do this, consider first that a key property of this foundation is that it:

I) allows informal, semi-formal, and formal specifications.

Property (I) considers that different heterogeneous system developers have dissimilar, yet successful, approaches for developing systems. In fact, different approaches might even be used within the same company depending on the particular system or function being developed. This fact can be understood by considering the development of a heterogeneous system as an optimization problem over stakeholder concerns such as cost, performance, availability, FuSa, security, etc. For example, formal (mathematical) specifications might be the most optimal solution when developing a complex safety-critical function; however, in other cases, informal specifications are more suitable.

Consider the intent of establishing a foundation with property (I) with the overall aim to maximize the support for the stringent RE advocated in FuSa standards. Notably, in order to provide maximum support for formal specifications, the fact is that the support must also be grounded in *formal* theory. That is, in order to employ formal techniques to verify a property such as completeness, the foundation must be based on an underlying formal theory that specifies exact mathematical conditions for when such a property holds or not. Actually, regardless if informal, semi-formal or formal specifications are used, a formal theory will enable more precise guidance to be given than an informal or semi-formal approach.

This means that non-formal approaches to RE, such as the one described in [1], although serving as a source for inspiration and ideas, does not serve as suitable theoretical starting point in literature.

Instead, consider formal RE theories such as the four-variable model [24], GORE models [25], the WRSPM reference model [26, 27] based on [28, 29], and more recent work [30, 31]. These theories have different aims with their formalizations, e.g. the work [24] seeks to formalize the content of existing system documents and [27] considers separating the needs of a user and supplier of a system. While having different aims, each of these theories lands in the same principal relation, namely that a system S satisfies a requirement R if:

$$A \wedge S \Rightarrow R \tag{1}$$

where A expresses assumptions on the environment of S. Notably, relation (1) is not explicitly mentioned in [30, 31] since satisfaction is defined for a set of systems instead of for a single system; rewriting this definition for a single system and the relation (1) will appear.

Notably, while introduced in the context of RE in [24, 29], the relation (1) can be traced back to much earlier work [32–34] in the area of compositional verification/reasoning [13, 14, 35]. In fact, relation (1) embodies the essential compositional characteristic of verifying in isolation that S satisfies R by assuming that an environment of S will fulfill A. In many compositional theories, the *assumptions* A and the requirement R are contained in a specification called a *contract* for S where R is typically denoted G, called a *guarantee*.

A main difference between formal compositional theories and the formal RE theories [24–27, 29], is that the former recognizes the need for establishing relation (1) at several levels and explicitly addresses this need in theory; on the other hand, in [24–27, 29], relation (1) is only considered for the case where R is a top-level requirement. In fact, as mentioned in [36], the theories [24, 26, 27, 29] do not have any explicit support for hierarchical structuring of requirements and in GORE models [25], requirements are not structured with respect to the architecture of a system. Furthermore, in contrast to compositional theories, the work in [30, 31] is not compositional and does not provide explicit support for ensuring completeness.

As described in Section 1, the fact is that FuSa standards such as ISO 26262 do advocate that requirements are structured in an hierarchical manner where completeness is ensured and that requirements are allocated to architecture elements at each level. Considering this fact and the previously mentioned limitations of formal RE theories, it holds that compositional theories are indeed more suitable than formal RE theories as a starting point for developing a foundation for supporting the stringent RE advocated in FuSa standards ; in fact, taking into account also the previously mentioned benefits from using compositional verification and that requirements are indeed structured in a compositional manner in FuSa standards, compositional theories provide a solid starting point in literature.

Therefore, the foundation proposed in this thesis has its starting point in existing literature on formal compositional theories (See Section 7 in Paper A for a survey). Furthermore, the underlying theory for the proposed foundation mainly draws inspiration from compositional *contract* theories. Actually, the underlying theory of the foundation proposed in this thesis *is* a compositional contract theory.

The principal reason for choosing to establish a compositional contract theory as underlying theory is that, in contrast to other compositional specification theories, contract theories strictly follow the principle of separation

of concerns/responsibilities [37]. Separating responsibilities between developers of parts is an important property in a complex development environment of heterogeneous systems, in particular in distributed OEM (Original Equipment Manufacturer)/supplier chains [38]. Moreover, this choice is actually non-restricting since *"any reasonably complete contract theory can be used as a specification theory"* [37].

Now that the starting point of the proposed foundation of this thesis has been properly described and motivated, Section 3.2, which now follows, will describe the foundation and its novelties in more detail.

3.2 Contract-Based Foundation for Specifying Heterogeneous Systems

As previously mentioned, this thesis presents a foundation, consisting of an underlying theory for specifying heterogeneous systems, complemented with principles and methods that describe how to apply the theory in order to support the stringent RE advocated in FuSa standards.

3.2.1 Key Properties of Foundation

Section 3.1 described one key property of this foundation, i.e. property (I). Two other key properties of the foundation are that it:

II) can support any design flow, e.g. top-down or bottom-up flow; and

III) supports specification of any part of a heterogeneous systems.

Property (II) caters to the same need as property (I), i.e. different heterogeneous system 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. This means that a formal approach such as Event-B [19], while providing mature formal development methodology, is not suitable as an overall solution since it considers a *top-down* design flow.

Note that the fact that the foundation can support any design flow, does not imply that the foundation does not apply in a context where a particular design flow is enforced. Rather, it means that the choice to enforce a particular design flow is left to a developer who wants to use the foundation.

Note also that the degree of which automation, guidance- and feedback-support can be provided, does vary depending on a specific design flow and level of formalization. This variation is dependent mostly on the level of formalization used in specifications; it is possible to provide support even if specifications are informal; however, a higher level of formality enables more support than a lower level.

As previously mentioned, the overall need that the proposed foundation addresses is to support stringent RE in *the domain of heterogeneous systems*, composed of SW, HW, and physical (mechanical, electrical, etc.) parts. Property (III) expresses a necessary property for addressing such a need, i.e. the foundation does not provide support only for specifying SW, but also for specifying HW and physical parts.

In order for a formal compositional theory to serve as underlying theory of a foundation with property (III), the compositional theory must be sufficiently expressive to support specifying SW, HW, and physical parts, typically modeled using different formalisms encompassing both continuous and discrete time dynamics. Not only that, but such a compositional theory must also be *unified* for heterogeneous systems, meaning that it should either embody a formalism that applies regardless of sub-domain or at least provide a means for mapping dissimilar formalisms to each other as done in [39]. This is needed in order to reason about completeness between requirement levels where the requirements specify properties of parts from different sub-domains.

Hence, while the proposed foundation has its general starting point in existing literature on compositional theories, this thesis focuses more on compositional theories that have the aim of being unified for heterogeneous system.

3.2.2 Research Methodology for Obtaining and Validating Foundation and its Industrial Impact

The underlying motivation for initiating this thesis work can be traced back to a master thesis written in 2011 by the author of this doctoral thesis and a fellow student named Julius Reineck. This master thesis work was performed at the automotive company Scania and involved a gap analysis between ISO 26262 and the product development process of Scania. In this gap analysis, it was determined that the greatest overall challenges in achieving compliance with this standard, are those that were also described

in Section 2.

With the overall aim to find solutions to address these challenges, a collaboration project called ESPRESSO was initiated between KTH and Scania. The author of this thesis was assigned as the main person to realize this aim by performing research in an industrial setting with direct feedback from engineers. The work presented in Papers A-D in this thesis is a result from such a tight collaboration with industry.

More specifically, this collaboration involved, but was not limited to, efforts such as validating industrial case studies, discussions/workshops with engineers, and having them apply principles and methods in practice. During the thesis work, the foundation in this thesis has been validated by being continuously trialed in such efforts and subsequently modified based on the gathered insights and results. While most of the details concerning these collaborative efforts cannot be disclosed due to confidentiality reasons, the insights and results from the efforts are indeed incorporated in the proposed foundation. In addition, one of the main end results from this collaboration is presented in Paper D that presents a specification tool implementing the foundation in the context of a tool chain at Scania. This industrial case-study implementation is also an example of the industrial impact of the foundation.

Another concrete example of the industrial impact of the proposed foundation, is the fact that an internal Scania standard, which is based on the foundation in this thesis, has been established to serve as a framework for systems development and achieving FuSa at Scania. Furthermore, the collaboration between KTH and Scania has resulted in that several new collaborative projects, which will build on the proposed foundation, have been initiated. These impacts manifest the fruitfulness of this collaboration as a whole and also the acknowledgment of the foundation at Scania.

The foundation is presented in Papers A-C included in this thesis. Each of these papers, and also Paper D, is structured as a logical argument:

1) an industrial need related to the overall challenges described in Section 2 is highlighted;

2) it is motivated (using mathematics if possible) why no current approach can address such a need;

3) the main contribution of the paper is claimed to address such a need;

4) this claim is shown to be true (typically using mathematics).

The main contribution addresses such a need on a *fundamental* level in accordance with properties (I)-(III). That is, the need is addressed on a level independent of level of formalization, design flow, and specific formalisms. Thus, when implementing the main contributions in Papers A-D, already existing approaches, concrete formalisms, tools, and techniques can be used.

Considering the proposed foundation as a whole, this means that aspects such as scalability, usability, etc. are abstracted and are relevant only when deciding on a specific way of implementing the foundation. Thus, comparing the foundation with more specific approaches with respect to such particular aspects is not needed considering that the foundation generalizes these approaches on a fundamental level.

As previously mentioned, the foundation as a whole has been continuously validated through collaborative efforts with industry; this includes the validation of the highlighted industrial need for each paper mentioned in argument point (1). Thus, in accordance with the argument structure (1)-(4), the industrial relevance of the main contribution of each Papers A-D is also well motivated. The contributions in Papers A-C are verified directly through formal reasoning and in particular, through mathematical theorems. The industrial applicability of concepts in Papers A-B has also been directly validated in a major industrial case study that is appended to Paper B in this thesis. Papers A-B are also indirectly validated by the fact that their underlying theory generalizes previously validated theories.

Paper D validates the entire foundation as a whole by showing that if a tool enforces the foundation, then the tool will provide feedback and guidance for the stringent RE advocated in FuSa standards. This is shown through formal reasoning, but also through the previously mentioned industrial case study where such a tool was implemented. Paper D also illustrates one of the many applications of the contributions presented in Paper C.

3.2.3 Contributions and Summary of Included Papers

As previously mentioned, the proposed foundation is presented and validated in Papers A-D included in this thesis. In short:

- Paper A presents a compositional contracts theory that is unified for heterogeneous systems and that caters to limitations in previous compositional contracts theories;

- Paper B extends this theory to also be applicable in a safety-critical context and presents methods for applying the theory to support stringent RE;
- Paper C describes principles for architecture modeling and specification of C programs using the theory described in Papers A and B; and
- Paper D validates the foundation in Paper A-C by describing how the proposed foundation can be implemented in an industrial setting.

The rest of this section describes the contributions of Papers A-D and their relations to each other in more detail.

Paper A: Conditions of Contracts for Separating Responsibilities in Heterogeneous Systems As previously presented, Paper A presents a compositional contracts theory, unified for heterogeneous systems. Notably, prior to Paper A, there already existed theories with similar scopes. The reason for establishing a new theory was that two major limitations were observed in these previous theories.

The first limitation is that conditions in previous compositional contract theories for fully separating responsibilities between developers of parts, are based on the assumption that interfaces of parts are always partitioned into inputs and outputs. However, consider typical models of the parts of a heterogeneous system, including not only SW, but also physical parts (mechanical, electrical, etc.). In addition to *causal* models where inputs and outputs are specified, there would also be *acausal models* [10] where the causality of the ports is unspecified, which is common in e.g. Modelica [40, 41]. Thus, this previously mentioned assumption is not generally valid in the domain of heterogeneous systems, which also implies that previous conditions for separating responsibilities are generally not applicable.

The second limitation is that in previous compositional contract theories, a contract for a part is limited to be expressed only over the port variables that constitute the interface of the part. As described thoroughly in Paper A, this limitation needs to be relaxed in order to properly express safety requirements and requirements in general.

As the main contribution, Paper A describes at a fundamental level how both these two limitations can be relaxed. However, not only that, but as a second contribution, Paper A also introduces revised definitions of contract

properties consistency and compatibility under the relaxation of these two limitations.

As a third contribution, in the context of a specific environment of a part, Paper A shows that there exists a necessary and sufficient set of port variables for expressing the assumptions and the guarantee of a contract for the part. Thus, without loss of expressiveness, contracts can be restricted to only be expressed over such sets to support specifying contracts that do not contain redundant port variable references.

As a fourth and final contribution, as a basis for structuring a set of contracts for compositional verification, a graph, called a *contracts composition structure*, is introduced. Based on a contracts composition structure, compositional verification conditions are also presented.

Paper B: Extending Contract Theory with SILs As previously described, Paper A presents a theory which addresses limitations in similar theories presented prior to Paper A. While comprehensive in many theoretical aspects, considering the overall aim of this thesis to support the stringent RE advocated in FuSa standards, Paper A is lacking in two aspects:

i) it is not explained thoroughly how contracts are related to commonly used concepts in RE and thus, it is unclear how to properly apply the theory to support stringent RE; and

ii) the relation between contracts and SILs is not described.

Paper B caters to these two lacking aspects (i) and (ii) by extending the theory in Paper A, recasting it in terms of RE concepts, and describing how to apply it to support stringent RE.

More specifically, considering aspect (i), Paper B introduces a graph-based method for hierarchical structuring of *requirements* with respect to the system *architecture* by considering a formal interpretation of completeness. This graph, called *contract structure* builds on the concept of a contract composition structure as described in Paper A and supports the individual tracing of requirements at a lower level back to their top-level requirement. This individual tracing of requirements is needed to comply with FuSa standards where the assignment of SILs to lower level requirements is determined based on their individual tracing to higher-level requirements.

More specifically, in FuSa standards such as IEC 61508 [4] and ISO 26262 [3], SILs or Automotive SILs (ASILs) as called in ISO 26262, are

assigned to top-level safety requirements. The higher the SIL is, the greater the amount of *safety measures*, i.e. activities or technical solutions that increase the reliability of a system, needs to be applied, in order to mitigate the occurrence of dangerous system failures. 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.

Consider that a requirement for a sub-system is assigned with a lower SIL than what is actually needed for the system to be sufficiently reliable in fulfilling a top-level safety requirement. This means that even if the sub-system is sufficiently reliable in fulfilling its requirement, it cannot be guaranteed that a tolerable level of risk is met. It is therefore imperative that the task of tracing requirements and assigning them SILs is done in a correct manner.

To support industry in completing such a rigorous task, based on a contract structure, the main contribution of Paper B also addresses aspect (ii) by presenting definitions that specify exact conditions under which SIL inheritance and decomposition can be applied when using contracts to structure requirements.

Paper C: Formal Architecture Modeling of Sequential Non-Recursive C Programs As previously mentioned, by relying on the theory in Paper A, Paper B presents a graph-based method for structuring requirements with respect to an architecture. This architecture is a hierarchy of elements with well-defined interfaces consisting of port variables.

In accordance with this method, requirements should at all levels be allocated to architecture elements, including SW requirements, which means that an architecture of SW is needed. This method is in accordance with the FuSa standard ISO 26262, where SW safety requirements must be allocated to SW elements and where it is stated that *"steps of the integration and testing of the software elements correspond directly to the hierarchical architecture of the software"* [3].

Notably, concerning architecture models, it is stated in [42] that *"to*

manage the complexity of developing, maintaining, and evolving a critical software-intensive system, its architecture description must be accurately and traceably linked to its implementation". In practice, in order to create architecture models that serve as *accurate* descriptions of systems, formal support is needed.

Considering formal support for describing and specifying systems as an hierarchy of elements with well-defined interfaces, for physical architectures, there are several mature formalisms such as Modelica [40]. There are also several formalisms for formal *specification* of SW architectures, e.g. Simulink [43] where SW can be generated from architecture models. However, for SW in general, there are not many approaches for formally *describing* already implemented SW. In particular, while being one of the most popular programming languages [44], prior to Paper C, formal support for describing the architecture of an implemented C program was missing.

The main contribution of Paper C establishes a *unique* mapping from the domain of sequential non-recursive C programs to the formal definition of an architecture as presented in Paper B. This mapping is fundamental and can be used both for describing and specifying architectures of C programs, and enables expressing and structuring requirements for C programs in a stringent manner.

As an additional contribution, considering that *manually* creating architecture models is an error-prone task in general [45], a study on the feasibility to automatically extract the proposed architecture model from a C program, is presented. Not only does the study indicate the potential of automatically extracting the proposed architecture model, but it also highlights specific C program constructs that are, in practice, difficult to extract from the C program code. Hence, the study also serves as a guideline for what C program constructs to avoid, in order to be able to generate architecture models in general, and architecture models of the proposed type, in particular.

Paper D: Providing Tool Support for Specifying Safety-Critical Systems by Enforcing Syntactic Contract Conditions Paper D summarizes, validates, and adds to the contributions in Papers A-C.

Paper D starts by explaining that a contract structure, as introduced in Paper B, sets a basis for achieving the stringent RE effort advocated by FuSa standards. Paper D then shows that establishing a contract structure with the intent of achieving completeness, consists of the tasks of specifying:

1) allocation of requirements to architecture elements;

2) architecture element interfaces consisting of port variables;

3) requirements; and

4) trace links between requirements.

Paper D also recognizes the fact that all these tasks have formal interpretations in the compositional contracts theory described in Paper A and its extension in Paper B. Paper D capitalizes on this fact by considering the support that can be provided for tasks (1)-(4) by having a tool that enforces these conditions.

Notably, in order to fully enforce all of these conditions, requirements and architecture need to be represented formally. Despite the fact that formal representations have several advantages over non-formal ones, formal languages are difficult to use by non-experts [46] and in industrial practice, *"overcoming the burden of formalization is a major challenge"* [47].

Considering this, Paper D first identifies the conditions of the formal interpretations of tasks (1)-(4) where these conditions can be enforced even when architectures and requirements are not represented formally; these necessary conditions are referred to as *syntactic conditions*. As the main contribution, Paper D shows that a specification tool will provide *feedback*- and *guidance*-driven support for tasks (1)-(4) by enforcing these syntactic conditions.

As a second contribution and validation of the main contribution, it is described how the proposed feedback- and guidance-driven support can be implemented in an industrial setting. This is achieved by describing the design and implementation solutions used for realizing this support in a validating industrial case study performed at Scania – a global heavy trucks manufacturer located in Sweden.

A key necessary concept for realizing this support in practice is shown to be Linked Data [48], which enables *formal referencing* in between specifications and to architecture data; formal referencing is necessary since the syntactic conditions are indeed formal, and thus, the conditions cannot be properly evaluated if it is unclear what data a reference points to in a specification. However, not only is this use of formal references a key enabler for realizing the main contribution in practice, but the use of these formal references also allows enforcing data in specifications to automatically be

consistent with architecture data, thus, reducing the error-prone and manual work of updating data manually. In fact, if combined with architecture recovery techniques as described in Paper C, consistency can even be enforced between specifications and implementation data.

All in all, Paper D justifies the previously stated claim that the foundation presented in this thesis is indeed suitable as a base for implementing tool support that reduces the manual effort to fill the gap described in Section 1.

References

[1] M. E. C. Hull, K. Jackson, and J. Dick, Eds., *Requirements Engineering, Third Edition*. Springer, 2011.

[2] B. H. C. Cheng and J. M. Atlee, "Research directions in requirements engineering," in *Future of Software Engineering, 2007. FOSE '07*, May 2007, pp. 285–303.

[3] ISO 26262, "Road vehicles-Functional safety," Geneva, Switzerland, 2011.

[4] IEC 61508, "Functional safety of electrical/electronic/programmable electronic safety-related systems," 2010.

[5] EN 50128, "Railway applications - communication, signalling and processing systems - software for railway control and protection systems," 2011.

[6] Def Stan 00-56, "Safety management requirements for defence systems," 2007.

[7] M. Rausand and A. Høyland, *System Reliability Theory: Models, Statistical Methods, and Applications*, ser. Wiley Series in Probability and Statistics - Applied Probability and Statistics Section. Wiley, 2004. [Online]. Available: https://books.google.se/books?id=gkUWz9AA-QEC

[8] T. Henzinger and J. Sifakis, "The Discipline of Embedded Systems Design," *Computer*, vol. 40, no. 10, pp. 32–40, Oct 2007.

[9] E. Lee, "Cyber Physical Systems: Design Challenges," in *Object Oriented Real-Time Distributed Computing (ISORC), 11th IEEE Int. Symp. on*, 2008, pp. 363–369.

[10] P. Fritzson, *Introduction to modeling and simulation of technical and physical systems with Modelica*. John Wiley & Sons, 2011.

[11] ISO/IEC/IEEE 42010, "System and software eng. - Architecture description," Geneva, Switzerland, 2011.

[12] J. Cleland-Huang, O. Gotel, and A. Zisman, *Software and Systems Traceability*. Springer Publishing Company, Incorporated, 2012.

[13] W. de Roever, H. Langmaack, and A. Pnueli, *Compositionality: The Significant Difference*. Springer, 1998.

[14] J. Hooman and W. P. de Roever, "The Quest Goes on: A Survey of Proofsystems for Partial Correctness of CSP," in *Current Trends in Concurrency, Overviews and Tutorials*. Springer Berlin Heidelberg, 1986, pp. 343–395. [Online]. Available: http://dx.doi.org/10.1007/BFb0027044

[15] J.-V. Millo, S. Ramesh, S. Krishna, and G. Narwane, "Compositional Verification of Software Product Lines," in *Integrated Formal Methods*, ser. Lecture Notes in Computer Science, E. Johnsen and L. Petre, Eds. Springer Berlin Heidelberg, 2013, vol. 7940, pp. 109–123.

[16] I. Schaefer, D. Gurov, and S. Soleimanifard, "Compositional Algorithmic Verification of Software Product Lines," in *Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, Eds. Springer Berlin Heidelberg, 2012, vol. 6957, pp. 184–203. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25271-6_10

[17] M. Nyberg, "Failure propagation modeling for safety analysis using causal bayesian networks," in *Control and Fault-Tolerant Systems (SysTol), 2013 Conference on*, Oct 2013, pp. 91–97.

[18] M. Nyberg and J. Westman, "Failure Propagation Modeling Based on Contracts Theory," in *Dependable Computing Conference (EDCC), 2015 Eleventh European*, Sept 2015, pp. 108–119.

[19] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *International journal on software tools for technology transfer*, vol. 12, no. 6, pp. 447–466, 2010.

[20] D. Firesmith, "Engineering safety requirements, safety constraints, and safety-critical requirements," *Journal of Object technology*, vol. 3, no. 3, pp. 27–42, 2004.

[21] N. G. Leveson, *Safeware: System Safety and Computers*. New York, NY, USA: ACM, 1995.

[22] A. M. Davis and D. Zowghi, "Good requirements practices are neither necessary nor sufficient," *Requirements Engineering*, vol. 11, no. 1, pp. 1–3, 2006.

[23] A. Sutcliffe and N. Maiden, "The domain theory for requirements engineering," *IEEE Transactions on Software Engineering*, vol. 24, no. 3, pp. 174–196, Mar 1998.

[24] D. L. Parnas, "Functional documents for computer systems," *Science of Computer Programming*, vol. 25, pp. 41–61, 1995.

[25] R. Darimont and A. Van Lamsweerde, "Formal refinement patterns for goal-driven requirements elaboration," in *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 6. ACM, 1996, pp. 179–190.

[26] J. G. Hall and L. Rapanotti, "A reference model for requirements engineering," in *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, Sept 2003, pp. 181–187.

[27] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave, "A reference model for requirements and specifications," *IEEE Software*, vol. 17, no. 3, pp. 37–43, 2000.

[28] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 1, pp. 1–30, Jan. 1997. [Online]. Available: http://doi.acm.org/10.1145/237432.237434

[29] M. Jackson, "The world and the machine," in *Proceedings of the 17th International Conference on Software Engineering*, ser. ICSE '95. New York, NY, USA: ACM, 1995, pp. 283–292. [Online]. Available: http://doi.acm.org/10.1145/225014.225041

[30] A. Goknil, I. Kurtev, and K. Van Den Berg, "Generation and Validation of Traces Between Requirements and Architecture Based on Formal Trace Semantics," *J. Syst. Softw.*, vol. 88, no. C, pp. 112–137, Feb. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.10.006

[31] A. Goknil, I. Kurtev, K. van den Berg, and J.-W. Veldhuis, "Semantics of trace relations in requirements models for consistency checking and inferencing," *Software & Systems Modeling*, vol. 10, no. 1, pp. 31–54, 2011. [Online]. Available: http://dx.doi.org/10.1007/s10270-009-0142-3

[32] R. W. Floyd, "Assigning meanings to programs," in *Mathematical Aspects of Computer Science*, ser. Proceedings of Symposia in Applied Mathematics, J. T. Schwartz, Ed., vol. 19. Providence, Rhode Island: American Mathematical Society, 1967, pp. 19–32.

[33] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[34] E. W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975. [Online]. Available: http://doi.acm.org/10.1145/360933.360975

[35] C. A. Furia, "A Compositional World," 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.102.3767

[36] M. W. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. P. Heimdahl, and S. Rayadurgam, "Your What Is My How: Iteration and Hierarchy in System Design," *IEEE Software*, vol. 30, no. 2, pp. 54–60, 2013.

[37] S. Bauer, A. David, R. Hennicker, K. Guldstrand Larsen, A. Legay, U. Nyman, and A. Wąsowski, "Moving from specifications to contracts in component-based design," in *Fundamental Approaches to Software Eng.*, ser. Lec. Notes in Computer Science, J. Lara and A. Zisman,

Eds. Springer Berlin Heidelberg, 2012, vol. 7212, pp. 43–58. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28872-2$_$3

[38] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple Viewpoint Contract-Based Specification and Design," in *Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Springer Berlin Heidelberg, 2008, vol. 5382, pp. 200–225. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92188-2_9

[39] M. W. Whalen, S. Rayadurgam, E. Ghassabani, A. Murugesan, O. Sokolsky, M. P. Heimdahl, and I. Lee, "Hierarchical multi-formalism proofs of cyber-physical systems," in *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on.* IEEE, 2015, pp. 90–95.

[40] P. Fritzson and V. Engelson, "Modelica - A unified object-oriented language for system modeling and simulation," in *ECOO'98 - Object-Oriented Programming*, ser. Lecture Notes in Computer Science, E. Jul, Ed. Springer Berlin Heidelberg, 1998, vol. 1445, pp. 67–90. [Online]. Available: http://dx.doi.org/10.1007/BFb0054087

[41] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach.* John Wiley & Sons, 2014.

[42] T. Mens, J. Magee, and B. Rumpe, "Evolving Software Architecture Descriptions of Critical Systems," *Computer*, vol. 43, no. 5, pp. 42–48, May 2010.

[43] J. B. Dabney and T. L. Harman, *Mastering simulink.* Pearson/Prentice Hall, 2004.

[44] S. Cass, "Top 10 programming languages," Jul. 2014. [Online]. Available: http://spectrum.ieee.org/computing/software/top-10-programming-languages

[45] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between source and high-level models," *SIGSOFT Softw. Eng. Notes*, vol. 20, no. 4, pp. 18–28, Oct. 1995. [Online]. Available: http://doi.acm.org/10.1145/222132.222136

[46] S. Farfeleder, T. Moser, A. Krall, T. Stålhane, H. Zojer, and C. Panis, "DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development," in *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2011 IEEE 14th International Symposium on*, April 2011, pp. 271–274.

[47] M. Böschen, R. Bogusch, A. Fraga, and C. Rudat, "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)*, ser. CEUR Workshop Proceedings. CEUR–WS, 2016, pp. 1–11. [Online]. Available: http://ceur--ws.org/Vol--1564/paper20.pdf

[48] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data-the story so far," *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, pp. 205–227, 2009.

Included Papers

Paper A

Conditions of Contracts for Separating Responsibilities in Heterogeneous Systems

Jonas Westman • Mattias Nyberg

Revised version of paper submitted to *Formal Methods of System Design*.

Abstract

A general, compositional, and component-based *contract* theory is proposed for modeling and specifying *heterogeneous systems*, characterized by consisting of parts from different domains, e.g. software, electrical and mechanical. Given a contract consisting of *assumptions* and a *guarantee*, clearly separated *conditions* on a component and its environment are presented where the conditions ensure that the guarantee is fulfilled - a *responsibility* assigned to the component, given that the environment fulfills the assumptions. To support both causal and acausal modeling, the conditions are applicable regardless if the set of ports of the component is partitioned into inputs and outputs or not, and hence fully support any scenario where components from different domains are to be integrated by solely relying on the information of a contract. An example of such a scenario of industrial relevance is explicitly considered, namely a scenario in a supply chain where the development of a component is outsourced. To facilitate the application of the theory in practice, necessary properties of contracts are also derived to serve as sanity checks of the conditions. Furthermore, based on a graph that represents a structuring of a hierarchy of contracts, sufficient conditions to achieve *compositionality* are presented.

1 Introduction

The notion of *contracts* was first introduced in [1] as a pair of pre- and post-conditions [2] to be used in formal specification of software (SW). In the more recent contract theory [3–5], developed within the European project SPEEDS [6], the use of contracts is extended from formal specification of SW to serving as a central systems engineering philosophy to support the design of *heterogeneous systems* [7–9]. Heterogeneous systems are characterized by consisting of parts from multiple domains, e.g. SW, mechanical, electrical, etc. The use of the theory [3–5] has been advocated in several contexts, e.g. Platform Based Design in [5], Model Based Design in [10], safety analyses in [11], virtual integration and testing in [12], and for structuring safety requirements in [13–15].

With the intent to be able to model and specify heterogeneous systems, the theory [3–5] considers a basic component-based modeling formalism that relies on a *"language-based abstraction where composition is by intersection.* [...] *No particular model of computation and communication is enforced, and*

continuous time dynamics such as those needed in physical system modeling is supported as well." [3]

More specifically, the theory [3–5] is centered around the notion of a *component* M with a set of *ports* and an *implementation*, also denoted M, which is an *assertion*, i.e. a set of value sequences over the ports. A *contract* C for the component M is a pair of assertions (A, G) where the *responsibility* that the *guarantee* G is fulfilled, is assigned to the component M, given that certain *assumptions* A are fulfilled - a responsibility of the environment of the component. This clear separation of responsibilities, embodied by a contract, is a principle for seamless integration of components, a primary concern in heterogeneous systems development, characterized by complex supply chains distributed over multiple organizations [3, 5].

To concretize the separation of responsibilities embodied by a contract in the theory [3–5], contract satisfiability conditions are presented. Given that *"M and C have the same ports"* [3], the component M *satisfies* C if it holds that $A \cap M \subseteq G$, which ensures that the composition of the component M and an environment component M_E where $M_E \subseteq A$, fulfills the guarantee, i.e. that $M_E \cap M \subseteq G$. Hence, given that the contract C is limited to the set of ports of M, to ensure that the guarantee G is fulfilled, conditions on the component M and its environment M_E can be separated as $A \cap M \subseteq G$ and $M_E \subseteq A$, respectively.

However, the guarantee is trivially fulfilled if it holds that $M_E \cap M = \emptyset$. This trivial solution is undesirable due to the fact that a component where $A \cap M = \emptyset$ (and in particular where $M = \emptyset$) trivially satisfies any given contract (A, G). Notably, the theory [3–5] does not explicitly address this undesirable case, but the theory does propose an approach to further separate responsibilities. This approach is to partition the sets of ports of the component M and its environment M_E into mirroring inputs and outputs, and enforcing the additional conditions that M and M_E are *receptive to their inputs*, i.e. that any values of the inputs are allowed by the implementations at any point in time. In the typical case, these additional conditions ensure that the non-trivial solution $M_E \cap M = \emptyset$ is avoided.

From these basic concepts of the theory [3–5], it can be observed that in order to enforce the conditions $A \cap M \subseteq G$, $M_E \subseteq A$, and that M and M_E are receptive to their inputs, where these conditions ensure both that the guarantee is fulfilled $M_E \cap M \subseteq G$ and that the trivial solution $M_E \cap M = \emptyset$ is avoided, the two following prerequisites apply:

a) the sets of ports of the component M and its environment M_E are partitioned into mirroring inputs and outputs; and

b) the contract $C = (A, G)$ is limited to the set of ports of the component M.

Hence, in order to ensure that the trivial solution $M_E \cap M = \emptyset$ is avoided, the theory [3–5] requires, as expressed in the prerequisite (a), that the causality of the ports are specified. However, considering typical models of the parts of a heterogeneous system, including not only SW, but also physical parts (mechanical, electrical, etc.), in addition to causal models, there would also be *acausal models* [16] where the causality of the ports is unspecified, which is common in e.g. *Modelica* [17, 18]. In fact, as stated in [16], when modeling physical parts, acausal models are well suited given that they reflect the physical structure of the parts and are also more reusable than causal models since the solution direction is not fixed. Hence, the prerequisite (a) expresses a limitation of the theory in [3–5] and highlights the need for more general conditions on a component and its environment where these conditions ensure that the guarantee is non-trivially fulfilled $\emptyset \neq M_E \cap M \subseteq G$, regardless if the prerequisite (a) holds or not.

Regarding the prerequisite (b), while the theory [3–5] only supports contracts that are limited to the sets of ports of components, there are at least two strong reasons why such a limitation needs to be relaxed in the context of specifying heterogenous systems.

The first reason is that such a relaxation allows assigning the responsibility that a global property is fulfilled, to a component, instead of just local properties expressed over its ports. The same need, but in the context of functions [19], has been identified in [20, 21]. An example where such a need is manifested in an industrial case study can be found in [22] where ModelicaML [23] is used to specify and verify requirements on a subsystem of a fuel management system where the requirements express the end-to-end functionality of the fuel management system in general. Another example can be found in [24] where SysML [25] is used to specify requirements on an engine knock controller and where the requirements allocated to the controller explicitly refer to parts, such as the piston, which is not a port of the controller but rather a port in its environment.

The second reason why contracts that are not limited to component ports are needed is due to the fact that, in the area of functional safety [26, 27], the

associated risk of a component, is assessed in the context of how it affects its environment. Hence, in order to properly express a safety specification for a component using a contract, there is a need to refer to parts in the environment that the component is to be deployed in [14]. For example, in the functional safety standard ISO 26262 [27], top-level safety requirements for an item, i.e. a system *within* the vehicle, are formulated in order to prevent or mitigate hazards, where the hazards

> *"shall be defined in terms of the conditions or behaviour that can be observed at the vehicle level"* [27].

Hence, in order for a contract to capture the fact that one overall responsibility of the item is to mitigate or prevent the hazard, which extends beyond the ports of the item, a contract that is not limited to the set of ports of the item, is needed. This can be observed in the industrial examples [13, 14], and also in [28], where safety requirements that are not limited to the sets of ports of components are necessarily used in order to properly express safety specifications for the components.

In the context of specifying heterogeneous systems, the two reasons above explain the importance of relaxing the limitation expressed in the prerequisite (b) such that assumptions and guarantees can be specified, not only over the ports of a component, but also over ports in the environment of the component. Considering this, and the previous stated fact that the prerequisite (a) also expresses a limitation of the theory [3–5], it can be concluded that there is a need for more general conditions on a component and its environment where these conditions ensure that the guarantee is non-trivially fulfilled $\emptyset \neq M_E \cap M \subseteq G$, regardless if the prerequisites (a) and (b) hold or not.

Contributions As the main contribution of the present paper, a set of clearly separated conditions on a component and its environment are established where the conditions ensure that the guarantee is non-trivially fulfilled. These conditions are, in contrast to the conditions presented in [3], applicable regardless if the prerequisites (a) and (b) hold or not. More specifically, the main contribution shows that in order for the relation $\emptyset \neq M_E \cap M \subseteq G$ to hold, the respective conditions on the component and the environment can be separated as:

i) $A \cap M \subseteq G$ and $A \cap G \subseteq M$; and

ii) $M_E \subseteq A$ and $M_E \cap G \neq \emptyset$,

where M is limited to constraining only the set of ports of the component, but where A and G are not. In fact, it is proven that the condition (i) is a necessary and sufficient condition on the component M to ensure that the relation $\emptyset \neq M_E \cap M \subseteq G$ holds for each environment M_E that is such that the condition (ii) holds. This means that the conditions (i) and (ii) fully support any scenario where the component and its environment are developed in complete isolation and where the only information that is shared, is the contract. An example of such a scenario of industrial relevance is explicitly considered throughout the paper, namely a scenario in a supply chain where the development of a component is outsourced.

A second contribution considers the fact that the relaxation of the limitation expressed in the prerequisite (b) leads to increased expressiveness with respect to how a contract can be specified. However, the increased expressiveness is not unlimited since there are necessary restrictions on the set of ports constrained by the assumptions and the guarantee of a contract in order for the conditions (i) and (ii) to hold. Therefore, to facilitate the specification of contracts in practice, conditions, called *scoping conditions*, are introduced to limit the set of ports that the assumptions and the guarantee of a contract can constrain; these scoping conditions ensure that the necessary restrictions are not violated without limiting expressiveness.

As a third contribution, considering that the contract properties *consistency* and *compatibility* are in [3, 5] defined under the limitations expressed in the prerequisites (a) and (b), generalized definitions of these properties are presented where these limitations are relaxed. More specifically, in [3, 5], the definitions of consistency and compatibility are based on the concept of receptivity to inputs, but have the overall aim to ensure that a contract is such that there in fact exist a component and an environment that are such their corresponding conditions, as defined in [3, 5], hold. In contrast to the definitions in [3, 5], which require the prerequisite (a) and (b) to hold, the definitions of consistency and compatibility in the present paper do not. Instead, consistency and compatibility are defined as necessary properties of conditions (i) and (ii); more specifically, consistency and compatibility are defined respectively as: if there exists a component and an environment that are such that their corresponding conditions (i) and (ii) hold.

As a fourth and final contribution, as a basis for structuring a contract C and a set of contracts $\{C_i\}_{i=1}^{N}$ in parallel to a composition M of a set of

components $\{M_i\}_{i=1}^N$ with the intent to establish that M is such that the condition (i) holds with respect to C, a graph, called a *contracts composition structure*, is introduced. Based on a contracts composition structure and in accordance with the principle of *compositionality* [29, 30], sufficient conditions are provided to ensure that: the composition M of $\{M_i\}_{i=1}^N$ is such that the condition (i) holds with respect to C if each component M_i is such that the condition (i) holds with respect to C_i. Hence, the fourth contribution supports an indirect way of establishing that M is such that the condition (i) holds with respect to C when a direct approach is not feasible due to e.g. the complexity of the composition.

Related Work Notably, in addition to [3–5], there are other general theories [31–57] for assume-guarantee reasoning. These theories and other related work will be described in more detail in Section 7, and the following will instead focus solely on arguing for the fact that the contributions of the present paper cannot be found in any of the theories [3–5, 31–57]. Since the second to fourth contribution are derived with the main contribution as a foundation, the main contribution alone will be considered as a basis of argumentation. Due to the fact that the part of the main contribution that specifically addresses the limitation expressed in the prerequisite (a), is rather specific to assume-guarantee theories with a satisfaction relation $A \cap M \subseteq G$, namely [4, 5, 32–35, 45, 53] and one of the instantiated theories in [31], only these will be considered when arguing for this part of the main contribution. The part of the main contribution regarding the fact that the conditions (i) and (ii) are applicable also under the relaxation of the limitation in the prerequisite (b), is compared to this aspect of expressiveness of other assume-guarantee theories, in general.

Considering [4, 5, 31–35, 45, 53], the theories [4, 31, 35] use the same type of approach as [3] to avoid the trivial solution $M_E \cap M \neq \emptyset$ and only [32] provides an alternative approach. In [32], a game-theoretic approach [58] is considered where the component and its environment take turns in changing the values (called states) of variables in a fixed set. To avoid the trivial solution $M_E \cap M \neq \emptyset$, the approach in [32] requires that each step (or transition) of a run also needs to be identified as an action taken by either the component or its environment. This means that the approach in [32] is not applicable in the case of conventional physical models (see e.g. Modelica [17, 18]) where interactions are modeled, not by an explicit separation of actions

taken by a component and its environment, but rather by equations that simultaneously constrain each other. In contrast to [32], the conditions (i) and (ii) in the present paper are indeed applicable without the need for identifying steps as component or environment actions, and are hence applicable for conventional physical models, as well as for models of any domain in general.

Despite the fact that contracts are limited to the sets of ports of components in [3], there are assume-guarantee theories that do not consider such a limitation, namely [32, 34, 38, 42, 45, 46, 51–53] and the meta theory in [31]. However, in contrast to the present paper, in the theories [34, 42, 45, 46, 51–53], the concept of ports is abstracted away by considering implementations, assumptions, and guarantees as formulas or abstract properties that are not bound to be specified over a certain structure such as a set of variables. In [32, 38], assumptions, guarantees, and implementations are all expressed over a fixed set of variables, which means that if a subset of the fixed set is associated as the set of ports of a component, it could be argued that contracts that are not limited to the set of ports of the component are supported. However, in contrast to the present paper, the association of a variable as a port of a component is only established informally since none of the theories [32, 38] incorporate a means to enforce the condition that the component implementation can only constrain its ports and no other variables in the fixed set. This does not only mean that the second contribution of the present paper is not derivable from the theories [31, 32, 34, 38, 42, 45, 46, 51–53], but also something more fundamental; these theories consider a looser notion of contract satisfiability where no conditions are enforced on the set of ports that a component can or cannot constrain. Hence, these theories do not capture the additional design conditions enforced on the component as expressed by its set of ports, which is a fundamental principle in many works, e.g. [37] and the present paper.

However, out of the theories [3–5, 33, 35–37, 39–41, 43, 44, 47–50, 54–57] where ports[1] are explicitly considered, none of these fully relax the limitation that assumptions and guarantees must be limited to component ports. In fact, the work in [37] is the only theory that partially relaxes this limitation by allowing assumptions that extend outside of the set of ports of

[1] A generalization of ports as defined in [3–5] is considered here where the concept of ports encompasses not just variables, but also *labels*, as long as they both are in a structure, e.g. a set, over which a component implementation is explicitly expressed. Hence, ports characterize inputs, outputs, messages, events, signals, actions, etc.

a component, but not guarantees. Thus, in contrast to [37], and also [3–5, 33, 35–37, 39–41, 43, 44, 47–50, 54–57], the present paper allows both assumptions and guarantees to be specified over ports that extend outside of the set of ports of a component and, thus, fully relaxes the limitation that contracts must be limited to component ports.

While none of the theories [3–5, 33, 35–37, 39–41, 43, 44, 47–50, 54–57] fully relax the limitation that contracts must be limited to component ports, it can be argued, considering the particular aspect of relaxing such a limitation, that these theories could serve as equivalent formalisms to the one in the present paper if the set of ports of a component are simply extended to include any port of the environment. These additional ports could further be labeled as "environment ports" to distinguish them from those inherent to the component. However, in addition to this, for these theories to serve as equivalent formalisms to the one in the present paper, there would also be a need to enforce the condition that the component implementation can only constrain the subset of its ports that are not labeled as environment ports. Notably, the theories [3–5, 33, 35–37, 39–41, 43, 44, 47–50, 54–57] do not support a straightforward way to enforce such a condition since, analogous to [32, 38], they do not incorporate a means to distinguish a port that an implementation constrains from one that it does not in the set over which the implementation is expressed. This can be contrasted with the present paper where such a condition is enforced at the foundation of the theory, influencing almost all definitions and theorems. Hence, considering the particular aspect of relaxing the limitation that contracts must be limited to component ports, it is clear that the theories [3–5, 33, 35–37, 39–41, 43, 44, 47–50, 54–57] do not serve as equivalent formalisms to the present paper and can neither be trivially extended to serve as such.

Organization of Paper Based on a theoretical foundation in Section 2, *contracts* are introduced in Section 3 along with the main contribution, namely the derivation of the conditions (i) and (ii). The basis is a scenario where a contract is used to outsource the development of a component. Considering the conditions (i) and (ii), Section 4 presents necessary restrictions on the set of variables over which assumptions and guarantees are specified. Generalized definitions of *consistency* and *compatibility* of a contract are established in Section 5 and Section 6 introduces *contracts composition structures* as a means to structure a hierarchy of contracts to achieve *compo-*

sitionality. Section 7 extends the related work in this section and Section 8 summarizes the paper and draws conclusions.

2 Assertions and Elements

This section establishes concepts for modeling a heterogeneous system and its parts, and to be able to derive the contributions concerning contracts and their properties as will be presented in Section 3, 4, 5 and 6. The concepts presented in this section mainly draws inspiration from the contract theory [3–5] developed in the European research projects SPEEDS [6]. Similarities between the concepts presented in this section and the ones presented in [3–5], as well as with other related work, are discussed briefly throughout the section and in more detail in Section 7.

2.1 Assertions and Runs

Let $X = \{x_1, \ldots, x_N\}$ be a non-empty set of variables. Consider a pair (x_i, ξ_i) consisting of a variable x_i and a trajectory $\xi_i = \{(t, x_i(t))\}_{t \in T}$ of values of x_i over a time-window $\emptyset \neq T \subseteq \mathbb{R}_{\geq 0}$. For example, Figure 1a shows a trajectory of values of the variable x_1. A set $\{(x_1, \xi_1), \ldots, (x_N, \xi_N)\}$ of such pairs with trajectories over the same time-window T, is called a *run* for X over T, denoted either $\omega_{X,T}$ or simply ω. For example, a run can be a *trace* [54, 59–61] or an *execution* as presented in [34]. As a more illustrative example, a run $\omega_{\{x_1,x_2,\},T}$ consisting of two pairs is shown in Figure 1b as a solid line where the trajectories of the pairs are also shown as two dashed lines. The trajectory of values of x_1 is the trajectory shown in Figure 1a and the other trajectory consists of values of variable x_2.

In the following, a universal set of variables Ξ will be assumed where Ω will denote the set of all possible runs for Ξ over each time-window $\emptyset \neq T \subseteq \mathbb{R}_{\geq 0}$. An *assertion* W is, a possibly empty, subset of Ω, i.e. $\mathsf{W} \subseteq \Omega$. This notion corresponds to similar definitions in theories [3–5, 14, 62]. However, these theories consider assertions as sets of runs for dissimilar sets of variables *local* to the assertions, rather than sets of runs for a *universal* set of variables as in the present paper. The choice of considering each run in an assertion for a universal set of variables, i.e. the set Ξ, is inspired by [32, 38] and allows to combine and compare assertions with the use of regular set operations (e.g. $\cup$, $\cap$) and set relations (e.g. $\subseteq$).

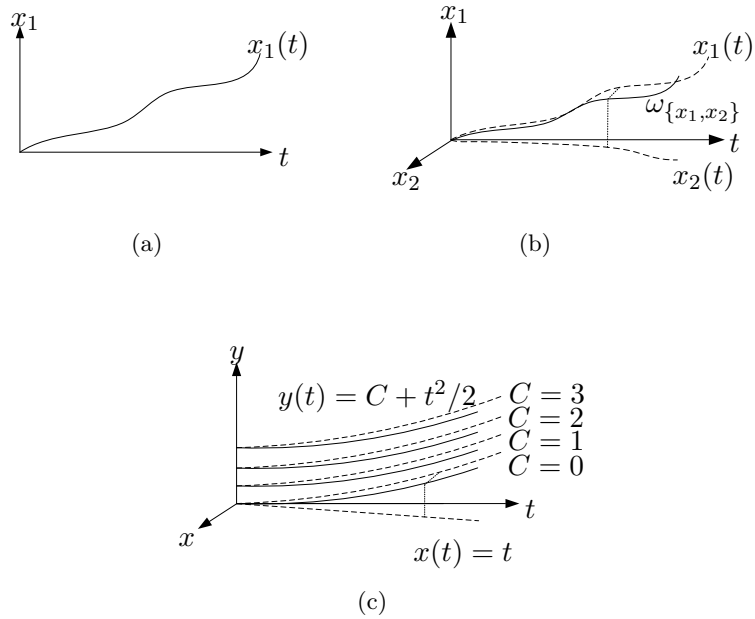


Figure 1. In (a), a trajectory of values of x_1 is shown. In (b), a run $\omega_{\{x_1,x_2\},T}$ is shown, consisting of two pairs containing the trajectory shown in (a) and another trajectory of values of x_2. In (c), a subset of the runs that are solutions to the differential equation $\frac{\mathrm{d}y}{\mathrm{d}t} = x(t)$, where $x(t) = t$ are shown.

Note that rather than explicitly declaring its set of runs, an assertion can be expressed through constraints, e.g. by equations, inequalities, or logical formulas. For example, an assertion W' expressed through the equation $u = v$, is the set of all runs in Ω where $u = v$ holds for each time point.

As a second example, consider that W'' is an assertion expressed through the first order differential equation

$$\frac{\mathrm{d}y}{\mathrm{d}t} = x(t), \text{ where } x(t) = t \text{ and } t \in \mathbb{R}_{\geq 0}.$$

Hence, the assertion W'' is the set of all possible runs for Ξ over $\mathbb{R}_{\geq 0}$ where the runs are the solutions to the differential equation. Figure 1c shows a subset of the solutions in the dimensions x, y, and t.

As a third example of how an assertion can be expressed, consider that W''' is an assertion expressed through the logic formula $a(t) = 0 \vee b(t) = 0$ where $t \in \{0, 1, \ldots, 10\}$ and where both variables a and b take values from $\{0, 1\}$. This means that the assertion W''' is the set of all possible runs for Ξ over the discrete time-window $\{0, 1, \ldots, 10\}$ where, for each time-point t, at least one of a and b has the value 0.

2.1.1 Projection of Assertions

Given an assertion W and a set of variables X, the *projection* [14, 31, 63] of W onto X, written $proj_X(\mathsf{W})$, is the set obtained when each pair that does not contain a variable $x \in X$ is removed from each run $\omega_{\Xi,T}$ in W, i.e. $proj_X(\mathsf{W})$ is equal to

$$\{\omega_{X,T} | \omega_{\Xi,T} \in \mathsf{W} \text{ and } \omega_{X,T} = \{(x, \xi) | (x, \xi) \in \omega_{\Xi,T} \text{ and } x \in X\}\} \ . \tag{1}$$

Note that $proj_X(\emptyset) = \emptyset$ and $proj_\emptyset(\mathsf{W} \neq \emptyset) = \{\emptyset\}$.

Using notation of relational algebra [64], it holds that $proj_X(\mathsf{W}) = \prod_X(\mathsf{W})$. Furthermore, the relation (1) corresponds to the definition of projection in [14, 31], while [63] defines projection as an operation on a single run instead of on a set of runs.

The *extended projection* of W onto X is denoted $\widehat{proj}_X(\mathsf{W})$ and is the assertion obtained if each run $\omega_{X,T}$ in $proj_X(\mathsf{W})$ is extended with all possible runs for $\Xi \setminus X$ over T. That is,

$$\widehat{proj}_X(\mathsf{W}) = \{\omega | \omega \in \Omega \text{ and } proj_X(\{\omega\}) \subseteq proj_X(\mathsf{W})\} \ . \tag{2}$$

Note that $\widehat{proj}_\emptyset(\mathsf{W}) = \Omega$ since it holds that $proj_\emptyset(\mathsf{W})$ is equal to $\{\emptyset\}$ and $proj_\emptyset(\{\omega\}) = \{\emptyset\}$ for each $\omega \in \Omega$.

The type of operation used for extending $proj_X(\mathsf{W})$ to the assertion $\widehat{proj}_X(\mathsf{W})$ is called *inverse projection* in [14, 31]. Furthermore, if W is expressed through a logical formula P, then the extended projection of W onto $\Xi \setminus \{x_1, \ldots, x_N\}$ corresponds to the notion of *port elimination* [3] or *variable hiding* [65, 66] through existential quantification, i.e. $\exists x_1, \ldots, x_N : P$.

As an example of projection and extended projection, consider an assertion expressed through the equation $y = x$, where the assertion will be

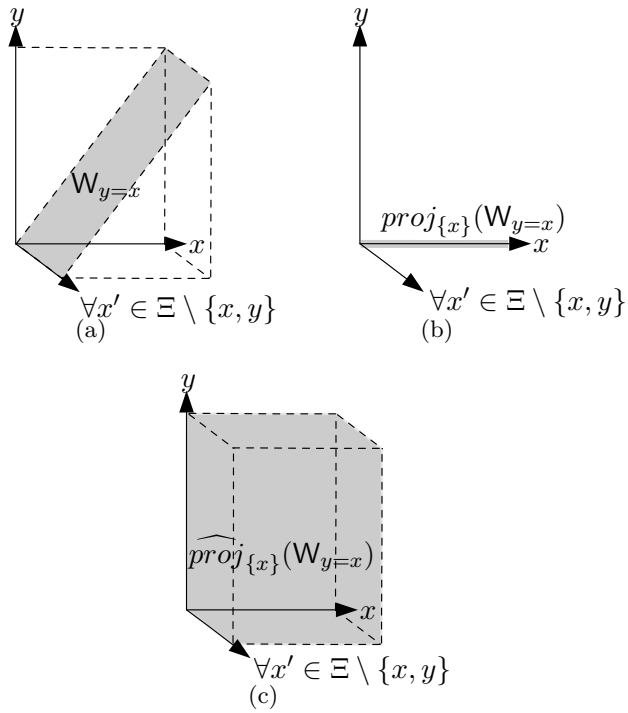


Figure 2. In a), b), and c), an assertion $\mathsf{W}_{y=x}$, its projection onto $\{x\}$, and its extended projection onto $\{x\}$ are shown, respectively.

denoted as $\mathsf{W}_{y=x}$ for convenience. The assertion $\mathsf{W}_{y=x}$ is shown in Figure 2a where the x'-axis can be the axis of any variable in $\Xi \setminus \{x, y\}$ and where only one single point in time is shown considering that $y = x$ is independent of time. The projection of $\mathsf{W}_{y=x}$ onto $\{x\}$, i.e. the set of runs $proj_{\{x\}}(\mathsf{W}_{y=x})$, is shown in Figure 2b. If each run $\omega_{\Xi,T}$ in $proj_{\{x\}}(\mathsf{W}_{y=x})$ is extended with all possible runs for $\Xi \setminus \{x\}$ over T, the extended projection of $\mathsf{W}_{y=x}$ onto $\{x\}$ is obtained. This assertion is shown in Figure 2c and is in fact the assertion Ω.

2.1.2 Variables Constrained by Assertions

As previously presented, assertions are sets of runs for the universal set of variables Ξ. However, as can be seen in previous examples, assertions can be expressed through constraints specified over a subset of Ξ. For example,

the equation $x = y$, through which the assertion $\mathsf{W}_{x=y}$ shown in Figure 2a is expressed, is specified simply over the set of variables $\{x, y\}$. For a given assertion, this section introduces a concept that distinguishes such as set of variables from the set of variables Ξ.

The concept of such a set is not presented in [3–5], but is, on the other hand, an essential concept in the present paper. This difference between [3–5] and the present paper can be explained by the fact that while each assertion consists of runs for the universal set of variables Ξ in the present paper, an assertion consists of runs for a set of variables local to the assertion in [3–5]. Considering that the use-cases in [3–5] show the clear intent that the set of local variables should be equal to the set of variables that are necessary and sufficient to express the assertion, no distinction between these two sets is required in [3–5]. In contrast, when considering assertions as defined in the present paper, while the runs of an assertion are for Ξ, the set of variables that are necessary and sufficient to express the assertion will, in the generic case, be a proper subset of Ξ, e.g. as exemplified with the assertion $\mathsf{W}_{x=y}$. Thus, to be able to directly refer to the subset of Ξ where this subset is necessary and sufficient to express an assertion, the concept of *the set of variables constrained by the assertion*, is introduced.

Definition 1 (Variables Constrained by Assertion)**.** *A variable x is constrained by an assertion* W if

$$\widehat{proj}_{\Xi\setminus\{x\}}(\mathsf{W}) \neq \mathsf{W} \ .$$

Let X_{W} denote *the set of variables constrained by* W. □

Notably, in accordance with Definition 1, to find the set X_{W}, i.e. the set of variables constrained by W, there is a need to iterate through each variable $x \in \Xi$ to determine whether or not it holds that $\widehat{proj}_{\Xi\setminus\{x\}}(\mathsf{W}) \neq \mathsf{W}$. The following proposition provides an alternative approach for finding X_{W} without the need for iterating over Ξ.

Proposition 1. *Given an assertion* W*, it holds that a set of variables X is equal to X_{W} if and only if each variable in X is constrained by* W *and* $\widehat{proj}_X(\mathsf{W}) = \mathsf{W}$.

The proof of Proposition 1 is found in Appendix I.

In accordance with Proposition 1, it holds that there exists a unique set of variables X constrained by W such that $\widehat{proj}_X(\mathsf{W}) = \mathsf{W}$. This unique set of variables is actually the set of variables X_{W} constrained by W.

As an example of how to use Proposition 1, consider the assertion $\mathsf{W}_{x=y}$. As shown in Figures 2a and 2c, the extended projection of $\mathsf{W}_{y=x}$ onto $\{x\}$ is not equal to $\mathsf{W}_{y=x}$, but rather a proper superset. The same holds for the extended projection of $\mathsf{W}_{y=x}$ onto $\{y\}$. This means that both the variables in the set $\{x, y\}$ are constrained by $\mathsf{W}_{y=x}$. However, not only that, since the equality $\widehat{proj}_{\{x,y\}}(\mathsf{W}_{x=y}) = \mathsf{W}_{x=y}$ implies, in accordance with Proposition 1, that $\{x, y\}$ is in fact also *the* set of variables constrained by $\mathsf{W}_{y=x}$.

2.2 Elements

In this section, the concept of an *element* is introduced. Elements correspond to Heterogeneous Rich Components (HRCs) [4, 67, 68], as used in the contract theory [3–5] of SPEEDS, in the sense that an element can represent any part of a heterogeneous system in general, such as a SW or physical part. However, an element can also serve as a *connector*, e.g. as described in Modelica [17, 18], or as a functional or logical design entity in general, e.g. as a SysML block [25]. The term element is in the present paper chosen over the term component due to the fact that the concept of elements also encompasses connectors, which are in Modelica [17, 18] and in theories such as [37], treated as separate entities from components.

Definition 2 (Element)**.** An *element* $\mathbb{E}$ is an ordered pair (X, B) where:

a) X is a non-empty set of variables, each called *a port variable*; and

b) B is an assertion, called the *behavior* of $\mathbb{E}$ and where the set of variables constrained by B is a subset of X. □

In the typical case, an element represents a real world entity where the port variables represent tangible quantities of the entity from the perspective of an external observer to the entity. The behavior of the element captures the static and dynamic constraints that the entity imposes on the quantities, independent of its surroundings.

Definition 2 is of a general type, which means that the conditions (a) and (b) hold regardless of the domain, e.g. mechanical, SW, etc., that is considered. However, in some domains, e.g. the SW domain, the set of port variables X of an element (X, B) is typically partitioned into inputs X^{in} and outputs X^{out}. In accordance with [3, 5], the partitioning of X into inputs and outputs enforces an additional condition on the behavior B, namely that

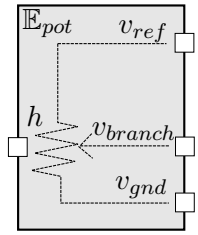


Figure 3. An element $\mathbb{E}_{pot} = (X_{pot}, \mathsf{B}_{pot})$, representing a potentiometer.

B *is receptive to* X^{in}, i.e. that $\widehat{proj}_{X_{in}}(\mathsf{B})$ is the set of all possible runs for X^{in} over each time-window in $\{T|\omega_{\Xi,T} \in \mathsf{B}\}$.

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 Figure 3 as a rectangle filled with gray and 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 expressed through the equation $h = \frac{v_{branch}-v_{gnd}}{v_{ref}-v_{gnd}}$.

2.3 Composition of Elements

This section describes how a set of elements $\{\mathbb{E}_1, \ldots, \mathbb{E}_N\}$ can be combined into a single element $\mathbb{E}$ - *a composition of* $\{\mathbb{E}_1, \ldots, \mathbb{E}_N\}$. In accordance with [3–5], 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. Sharing of port variables is also used in e.g. [56].

Prior to presenting a formal definition of composition of a set of elements, the concept is introduced by considering a set of elements representing the parts of a "Level Meter system" (LM-system) as shown in Figure 4. 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 Figure 3, a battery $\mathbb{E}_{bat}$, and a level meter $\mathbb{E}_{lMeter}$. The sharing of a port variable between elements is shown as either by a line connecting two or more boxes corresponding to

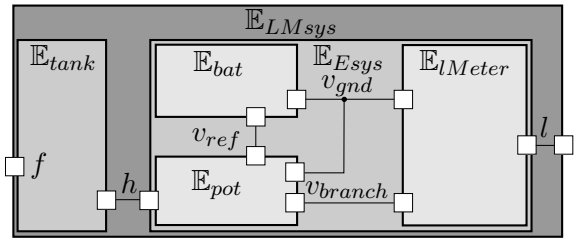


Figure 4. A set of elements representing a "Level Meter system" and its parts.

the same port variable or by the appearance of the same box on edges of several rectangles.

In the LM-system, the slider h 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 $\mathbb{E}_{bat}$, $\mathbb{E}_{lMeter}$, and $\mathbb{E}_{tank}$ are expressed through the equations $v_{ref} - v_{gnd} = 5V$, $l = \frac{v_{branch} - v_{gnd}}{5V}$, and $h = f$, respectively. The intersection $\mathsf{B}'_{LMsys} = \mathsf{B}_{bat} \cap \mathsf{B}_{Lmeter} \cap \mathsf{B}_{tank} \cap \mathsf{B}_{pot}$ captures the combined constraints expressed by the behaviors of $\mathbb{E}_{bat}$, $\mathbb{E}_{lMeter}$, $\mathbb{E}_{tank}$, and $\mathbb{E}_{pot}$. The assertion B'_{LMsys} is also the behavior of an element $(X'_{LMsys}, \mathsf{B}'_{LMsys})$, called *the composition of* $\{\mathbb{E}_{bat}, \mathbb{E}_{lMeter}, \mathbb{E}_{tank}, \mathbb{E}_{pot}\}$ where $X'_{LMsys} = X_{bat} \cup X_{Lmeter} \cup X_{tank} \cup X_{pot}$.

While B'_{LMsys} captures the combined constraints expressed by the behaviors of $\mathbb{E}_{bat}$, $\mathbb{E}_{lMeter}$, and $\mathbb{E}_{tank}$, and $\mathbb{E}_{pot}$, the set of port variables that is constrained by the assertion B'_{LMsys} is a proper *superset* of the set of port variables $X_{LMsys} = \{f, l\}$ of $\mathbb{E}_{LMsys}$ as shown in Figure 4. In accordance with Definition 2, this means that B'_{LMsys} cannot be the behavior of $\mathbb{E}_{LMsys}$. The behavior B_{LMsys} of $\mathbb{E}_{LMsys}$ is instead the extended projection of B'_{LMsys} onto $\{f, l\}$, which, in accordance with the relation (2), means that B_{LMsys} can be expressed through the equation $l = f$. The element $\mathbb{E}_{LMsys}$ is also called *the composition of* $\{\mathbb{E}_{bat}, \mathbb{E}_{lMeter}, \mathbb{E}_{tank}, \mathbb{E}_{pot}\}$ *onto* X_{LMsys}. The element $\mathbb{E}_{Esys}$ is the composition of $\{\mathbb{E}_{bat}, \mathbb{E}_{lMeter}, \mathbb{E}_{pot}\}$ onto $X_{Esys} = \{h, l\}$.

Now that the concept of element composition has been introduced, the formal definition follows.

Definition 3 (Composition of Elements (onto a Set of Variables))**.** The

composition of a set of elements $\{(X_1, \mathsf{B}_1), \ldots, (X_N, \mathsf{B}_N)\}$ *onto a non-empty set* $X \subseteq \bigcup_{i=1}^{N} X_i$, is the element $(X, \widehat{proj}_X(\bigcap_{i=1}^{N} \mathsf{B}_i))$. In the case where $X = \bigcup_{i=1}^{N} X_i$, the composition of $\{(X_1, \mathsf{B}_1), \ldots, (X_N, \mathsf{B}_N)\}$ onto X, is simply called the composition of $\{(X_1, \mathsf{B}_1), \ldots, (X_N, \mathsf{B}_N)\}$. □

In accordance with Definition 3 and as previously indicated, in the case where $X = \bigcup_{i=1}^{N} X_i$, the composition of $\{(X_1, \mathsf{B}_1), \ldots, (X_N, \mathsf{B}_N)\}$ onto X, is simply called the composition of $\{(X_1, \mathsf{B}_1), \ldots, (X_N, \mathsf{B}_N)\}$. This case is in accordance with the definition of composition of HRCs in [3–5].

In the case where $X \subset \bigcup_{i=1}^{N} X_i$, Definition 3 combines composition as defined in [3–5] with *port elimination* [3], also called *variable hiding* [65, 66]. As shown in the example in Figure 4, this case allows representing hierarchical systems. For example, in [69], the overall concept of elements and how they compose were used for representing C-programs as architecture models.

Given a set of elements $\mathcal{E}$, *the environment of an element* $\mathbb{E} \in \mathcal{E}$, denoted $\mathbb{E}_{Env_{\mathcal{E}}(\mathbb{E})}$, is the composition of $\mathcal{E} \setminus \{\mathbb{E}\}$. As an example of the environment of an element, given a subset $\mathcal{E}_{LMsys} = \{\mathbb{E}_{bat}, \mathbb{E}_{tank}, \mathbb{E}_{pot}, \mathbb{E}_{lMeter}\}$ of the elements shown in Figure 4, the environment $\mathbb{E}_{Env_{\mathcal{E}_{LMsys}}(\mathbb{E}_{lMeter})}$ of $\mathbb{E}_{lMeter}$ is the composition of $\{\mathbb{E}_{bat}, \mathbb{E}_{tank}, \mathbb{E}_{pot}\}$.

3 Conditions of Contracts for Separating Responsibilities

Based on the concepts presented in Section 2, this section introduces the concept of a *contract* containing assumptions and a guarantee, and also *conditions* that ensure that the guarantee is fulfilled.

3.1 Contracts

As mentioned in Section 1, the notion of contracts was first introduced in [1] as a pair of pre and post-conditions [2] to be used in formal specification of SW interfaces. In the recent contract theory [3–5], developed within SPEEDS [6], the use of contracts is extended from formal specification of SW to serving as a central philosophy in systems engineering to support the design of heterogeneous systems. One of the key challenges that triggered the extension of contracts is the increasingly complex development environ-

ment of heterogeneous systems, characterized by distributed OEM (Original Equipment Manufacturer)/supplier chains [3].

In the context of an OEM/supplier chain, in order for a global intended property to be fulfilled by a composition of a set of elements, the OEM needs to distribute the responsibilities of fulfilling local properties between different elements that are to be integrated into the composition. Considering that these elements are to be developed by different suppliers, clearly defined interfaces and the separation of responsibilities between the different elements are paramount in order to support seamless integration. A *contract* addresses such concerns by assigning the responsibility that a certain property is fulfilled, to an element in the form of a *guarantee*, given that certain *assumptions* are fulfilled - a responsibility of the *environment* of the element.

Although the discussion above focuses on the use of contracts for managing the complexity of OEM/supplier chains, the discussion can be generalized and is equally valid for any design context where clear separations of responsibilities are desired.

A contract $(\{\mathsf{A}_i\}_{i=1}^{N}, \mathsf{G}, X)$ is a specification for an element $\mathbb{E}$ with a set of port variables X, expressing the intent that the behavior of the element is such that the guarantee G is fulfilled, given a set of elements containing $\mathbb{E}$ where its environment fulfills the assumptions in $\{\mathsf{A}_i\}_{i=1}^{N}$.

Definition 4 (Contract)**.** A *contract* $\mathcal{C}$ is a tuple $(\mathscr{A}, \mathsf{G}, X)$, where

i) G is an assertion, called *guarantee*;

ii) $\mathscr{A}$ is a set of assertions $\{\mathsf{A}_i\}_{i=1}^{N}$ where each A_i is called an *assumption*; and

iii) X is a set of variables. □

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

As an illustrative example of a contract, let $(\{\mathsf{A}_{lMeter}\}, \mathsf{G}_{lMeter}, X_{lMeter})$ be a contract $\mathcal{C}_{lMeter}$ where the set of port variables constrained by A_{lMeter} and G_{lMeter} are shown as dashed lines in Figure 5. The guarantee G_{lMeter}, specified by the equation $l = f$, expresses the intent that the indicated level, displayed by the meter, corresponds to the level in the tank. In the context of the set of elements $\mathcal{E}_{LMsys} = \{\mathbb{E}_{bat}, \mathbb{E}_{tank}, \mathbb{E}_{pot}, \mathbb{E}_{lMeter}\}$, the responsibility that the guarantee G_{lMeter} is fulfilled, is assigned to $\mathbb{E}_{lMeter}$, but only if the voltage measured between v_{branch} and v_{gnd} maps to a specific level in the

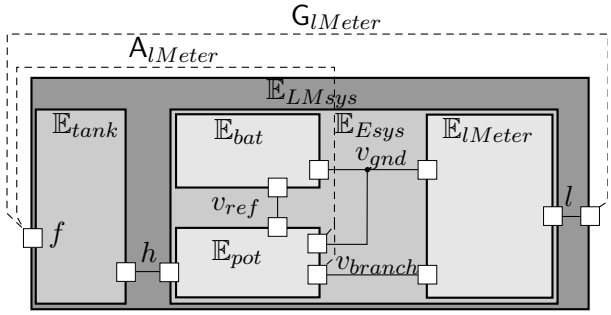


Figure 5. A set of elements representing a "Level Meter system" and its parts, and a contract $\mathcal{C}_{lMeter} = (\{\mathsf{A}_{lMeter}\}, \mathsf{G}_{lMeter}, X_{lMeter})$.

tank, i.e. only if the environment $\mathbb{E}_{Env_{\mathcal{E}_{LMsys}}(\mathbb{E}_{lMeter})}$ fulfills the assumption A_{lMeter}, specified by the equation $f = \frac{v_{branch} - v_{gnd}}{5V}$.

In general assume-guarantee theories [3, 4, 33, 35–37, 39–41, 43, 44, 47–50, 54–57] where ports are explicit, the contract $\mathcal{C}_{lMeter}$ is not supported since the assumption A_{lMeter} and the guarantee G_{lMeter} are not specified only over the set of ports of $\mathbb{E}_{lMeter}$, i.e. over the voltage connections v_{branch} and v_{gnd}, and the indicated level l. In contrast to these theories, the present paper supports specifying a contract for $\mathbb{E}_{lMeter}$ where the assumption A_{lMeter} and the guarantee G_{lMeter} constrain port variables that are not in the set of port variables of $\mathbb{E}_{lMeter}$ as exemplified in Figure 5.

The contract $\mathcal{C}_{lMeter}$ can for example be used in a scenario where an OEM develops the $\mathbb{E}_{lMeter}$ in-house while the development of the elements $\mathbb{E}_{bat}$, $\mathbb{E}_{pot}$, and $\mathbb{E}_{tank}$ are outsourced to suppliers. The responsibility that the overall intended functionality of the LM-system, as expressed by G_{lMeter}, is fulfilled, is assigned to $\mathbb{E}_{lMeter}$, meaning that the OEM does not only need to ensure the development of $\mathbb{E}_{lMeter}$, but also its successful integration with the elements $\mathbb{E}_{bat}$, $\mathbb{E}_{tank}$, and $\mathbb{E}_{pot}$ into the composition of $\mathcal{E}_{LMsys}$ that fulfills G_{lMeter}. A successful integration of the elements can be ensured by the OEM, given that the assumption A_{lMeter} is fulfilled. This is a responsibility of the environment $\mathbb{E}_{Env_{\mathcal{E}_{LMsys}}(\mathbb{E}_{lMeter})}$ that is to be developed by the suppliers.

3.2 Conditions on Element and Environment

Given a set of elements $\mathcal{E}$ and a contract $(\mathscr{A}, \mathsf{G}, X)$ for an element $\mathbb{E} = (X, \mathsf{B}) \in \mathcal{E}$, this section proposes the following respective conditions on the element $\mathbb{E}$ and on its environment $\mathbb{E}_{Env_{\mathcal{E}}(\mathbb{E})}$:

i) $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ and $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$; and

ii) $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$.

As will be shown in this section, these conditions ensure that *the guarantee is fulfilled*, which can expressed as

$$\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G} \, . \tag{3}$$

In fact, not only that, but the condition (i) on the element $\mathbb{E}$ actually ensures that the relation $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$ holds for *each set of elements* where the environment of $\mathbb{E}$ is such that the condition (ii) holds. Furthermore, the conditions (i) and (ii) also ensure that the trivial solution where $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} = \emptyset$, is avoided. That is, these conditions actually ensure that *the guarantee is non-trivially fulfilled*, i.e. that it holds that

$$\emptyset \neq \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G} \, . \tag{4}$$

In contrast to [3–5] where the trivial solution $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} = \emptyset$ is avoided only in the specific case where the set of ports X is partitioned into inputs and outputs, the conditions that will presented in this section, are applicable in general. This includes the case where the assumptions and the guarantee are not limited to constraining port variables in X; a case that is prohibited in [3–5].

In order to get a better understanding of when the conditions (i) and (ii) are needed, a scenario in the context of an OEM/supplier chain as previously presented, is examined. In the scenario, a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ is used to outsource the development of an element $\mathbb{E} = (X, \mathsf{B})$. Specifically, the scenario can be described in three phases:

I) a contract $\mathcal{C}$ is handed from the OEM to a supplier;

II) the supplier develops an element $\mathbb{E} = (X, \mathsf{B})$ that is handed to the OEM; and

III) the OEM integrates the element $\mathbb{E}$ with a set of elements $\{\mathbb{E}_i\}_{i=1}^N$ into the composition of $\mathcal{E} = \{\mathbb{E}\} \cup \{\mathbb{E}_i\}_{i=1}^N$.

As expressed in the phases (I-II), the development of the element $\mathbb{E}$ is guided only by the information available in the contract $\mathcal{C}$, i.e. without access to the environment $\mathbb{E}_{Env_{\mathcal{E}}(\mathbb{E})}$ of $\mathbb{E}$. Therefore, in order for the composition of $\mathcal{E}$ in the phase (III) to be such that the relation (4) holds with respect to $\mathcal{C}$, conditions must be enforced on the element $\mathbb{E}$ such that the relation (4) holds not just for the set $\mathcal{E}$, but rather for *each set of elements* containing $\mathbb{E}$ where the environment is such that certain conditions hold.

As will be shown in the following sections, the conditions (i) and (ii) are, in fact, instantiations of such conditions. The subset of the conditions (i) and (ii) that ensure that the relation $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$ holds will first be derived in Section 3.2.1, followed by the remaining subset that ensure that also the relation $\emptyset \neq \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B}$ holds in Section 3.2.2.

3.2.1 Conditions Ensuring Guarantee is Fulfilled

As previously mentioned, in the context of $\mathcal{E}$, the responsibility that the guarantee is fulfilled, is assigned to $\mathbb{E}$, given that *the environment of* $\mathbb{E}$ *fulfills the assumptions*. This means that it must hold that

$$\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}} \ . \tag{5}$$

Supposing that the relation (5) holds, it follows that the relation (3) holds if

$$\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G} \ . \tag{6}$$

Note that if $\mathscr{A} = \emptyset$, then the relation (6) simplifies to $\mathsf{B} \subseteq \mathsf{G}$. The condition (6) on the element has previously been identified in e.g. [3–5, 31, 32, 62].

The relation (6) is a *sufficient*, but not necessary condition for the relation (3) to hold considering a *specific* set of elements where the environment is such that the relation (5) holds. As expressed in the following proposition, the relation (6) is also a *necessary* condition in order for the relation (3) to hold for *each set of elements* where the environment is such that the relation (5) holds.

Proposition 2. *Consider a contract* $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ *and an element* $\mathbb{E} = (X, \mathsf{B})$. *It holds that* $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$, *if and only if* $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$ *for each set of elements* $\mathcal{E} \ni \mathbb{E}$ *where* $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$.

Proof. Consider a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and an element $\mathbb{E} = (X, \mathsf{B})$.

For the if-only case, assume that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$. Consider an arbitrary set of elements $\mathcal{E} \ni \mathbb{E}$ where it holds that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$. The relations $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ imply that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$. Since $\mathcal{E}$ was chosen arbitrarily, it means that the relation $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$ also holds for each set of elements $\mathcal{E} \ni \mathbb{E}$ where it holds that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$.

For the if case, assume that the relation $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$ holds for each set of elements $\mathcal{E} \ni \mathbb{E}$ where $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$. Assume that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \not\subseteq \mathsf{G}$, which will be shown to lead to a contradiction. Assume a set of elements $\mathcal{E} \ni \mathbb{E}$ where $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} = \mathsf{A}_{\mathscr{A}}$, which means that it follows that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} = \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$. This contradicts the fact that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \not\subseteq \mathsf{G}$, which means that it must rather hold that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$, which concludes the proof. □

Proposition 2 expresses that the relation (6) on the element $\mathbb{E}$ is a necessary and sufficient condition in order to obtain an element and its environment in the phase (III) such that the relation (3) holds in general, given that the relation (5) on the environment holds. However, the relation (3) trivially holds if the behavior of the composition of the element and the environment is empty, which would imply that the relation (4) does not hold. Therefore, the relation (4) does not follow from the relations (5) and (6), which means that additional conditions must be imposed on the environment and on the element in order to ensure that the trivial solution is avoided. In the following, these conditions will be examined.

3.2.2 Conditions Ensuring Non-Triviality

Consider a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and an element $\mathbb{E} = (X, \mathsf{B})$, and assume a set of elements $\mathcal{E}$ containing $\mathbb{E}$ such that its environment is such that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} = \emptyset$. Notably, if the behavior of the composition of the element and its environment is non-empty, i.e. if $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \neq \emptyset$, then it must follow that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \not\subseteq \mathsf{G}$ since none of the runs in $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})}$ are in G. Hence, in order for the relation (4) to hold, the environment must be such that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$. This insight is summarized in the following proposition.

Proposition 3. *Consider a contract* $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ *and a set of elements* $\mathcal{E}$ *containing an element* $\mathbb{E} = (X, \mathsf{B})$*. It holds that* $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$ *if* $\emptyset \neq \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$.

Proof. Consider a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and a set of elements $\mathcal{E}$ containing an element $\mathbb{E} = (X, \mathsf{B})$. Assume that $\emptyset \neq \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$. Intersecting both sides of this relation with $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})}$, yields $\emptyset \neq \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G}$. This implies that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$. □

Now that the necessary condition $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$ on the environment of $\mathbb{E}$ has been identified in order for the relation (4) to hold, a complementary sufficient condition on the element $\mathbb{E}$ is examined in order to ensure that the relation (4) holds.

Consider that the relation $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ on the element $\mathbb{E}$ holds. As shown in the Venn diagram in Figure 6a, if it holds that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$, it is possible that the relation (4) holds. However, as shown in Figure 6b, this is not true for all cases. In fact, since $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G}$ can simply consist of one run, and this run can possibly be any run in $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G}$, in order to ensure that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \neq \emptyset$, it must hold that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$ as shown in Figure 6c.

These insights are now summarized in the following theorem.

Theorem 1. *Consider a contract* $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ *and an element* $\mathbb{E} = (X, \mathsf{B})$ *where* $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$*. It holds that*

$$\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B},$$

if and only if $\emptyset \neq \mathsf{B} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{G}$ *for each set of elements* $\mathcal{E}$ *containing* $\mathbb{E}$ *where* $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$ *and* $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$*.*

To clarify Theorem 1, by using quantifiers, Theorem 1 can also be expressed as: given a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and an element $\mathbb{E} = (X, \mathsf{B})$ where $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$, it holds that

$$\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B} \iff$$

$$(\forall \mathcal{E} \ni \mathbb{E} : ((\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset \wedge \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}) \implies \emptyset \neq \mathsf{B} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{G})) .$$

Proof. Consider a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and an element $\mathbb{E} = (X, \mathsf{B})$ such that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$.

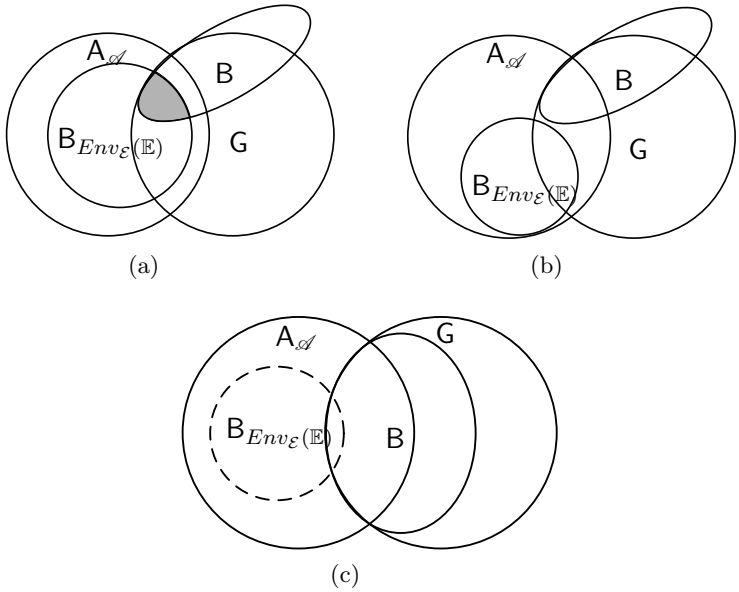


Figure 6. The Venn diagram in (a) shows a case where it holds that $\emptyset \neq \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$ and (b) shows a case where this does not hold. In (c), a Venn diagram is shown where it holds that $\emptyset \neq \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$ for each set $\mathcal{E}$ where $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$.

For the if-only part, assume that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$. Furthermore, consider a set of elements $\mathcal{E}$ containing $\mathbb{E}$ where $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$. The relations $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ imply that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \subseteq \mathsf{B}$. This and the fact that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$ imply that $\emptyset \neq \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \subseteq \mathsf{B}$. Intersecting both sides of this relation with $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})}$, yields $\emptyset \neq \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \subseteq \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B}$. This implies that $\mathsf{B}_{Env_{\mathscr{A}}(\mathbb{E})} \cap \mathsf{B} \neq \emptyset$. This and since the relations $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ imply that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$, it can be concluded that $\emptyset \neq \mathsf{B} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{G}$. Considering that $\mathcal{E}$ was chosen arbitrarily, it follows that the relation $\emptyset \neq \mathsf{B} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{G}$ also holds for each set of elements $\mathcal{E}$ containing $\mathbb{E}$ where $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$. This completes the if-only part of the proof.

For the if part, assume that for each set of elements $\mathcal{E}$ containing $\mathbb{E}$ where $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$, it follows that $\emptyset \neq \mathsf{B} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{G}$. Assume that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \not\subseteq \mathsf{B}$, which will be shown to lead to a contradiction.

This means that there exists a run ω such that $\omega \in \mathsf{A}_{\mathscr{A}} \cap \mathsf{G}$ and $\omega \notin \mathsf{B}$. Furthermore, assume that there exists a set of elements $\mathcal{E}$ containing $\mathbb{E}$ where $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} = \{\omega\}$. This and the fact that $\omega \in \mathsf{A}_{\mathscr{A}} \cap \mathsf{G}$ imply that both the relations $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$ hold. As was assumed, this means that it follows that $\emptyset \neq \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \subseteq \mathsf{G}$. However, this is a contradiction since the fact that $\omega \notin \mathsf{B}$ implies that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} = \emptyset$. It follows that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \not\subseteq \mathsf{B}$ cannot be true, which means that it must hold that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$, which concludes the proof. □

Given that the relation $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ on the element $\mathbb{E}$ holds, Theorem 1 expresses a necessary and sufficient condition on the element $\mathbb{E}$ such that, for each set of elements $\mathcal{E} \ni \mathbb{E}$ where $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$, it holds that the composition of $\mathcal{E}$ non-trivially fulfills G. The condition expressed in Theorem 1 holds regardless of the considered domain, e.g. SW, electrical, mechanical, etc, and does not require the set of port variables of the element to be partitioned into inputs and outputs.

Theorem 1 quantifies over sets of elements, and, thus, also over various environments of the element $\mathbb{E}$. To clarify the respective conditions on the element $\mathbb{E}$ and its environment $\mathbb{E}_{Env_{\mathcal{E}}(\mathbb{E})}$ in a *specific set of elements*, the following corollary simplifies Theorem 1 by removing the quantification over sets of elements.

Corollary 1. *Given a contract* $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ *and a set of elements* $\mathcal{E}$ *containing an element* $\mathbb{E} = (X, \mathsf{B})$*, it holds that* $\emptyset \neq \mathsf{B} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{G}$ *if both the following conditions hold:*

i) the element $\mathbb{E}$ *is such that*

$$\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}, \text{ and} \tag{7}$$
$$\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}; \tag{8}$$

ii) the environment $\mathbb{E}_{Env_{\mathcal{E}}(\mathbb{E})}$ *of* $\mathbb{E}$ *is such that*

$$\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}, \text{ and} \tag{9}$$
$$\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset\ . \tag{10}$$

Proof. Trivially follows from Theorem 1. □

In the context of the scenario presented in the beginning of this section, Corollary 1 specifies the conditions that the OEM and the supplier need to meet in order to ensure that the integration of $\mathbb{E}$ with the elements in $\{\mathbb{E}_i\}_{i=1}^{N}$ in the phase (III) results in that the guarantee is non-trivially fulfilled by the composition of $\{\mathbb{E}\} \cup \{\mathbb{E}_i\}_{i=1}^{N}$, i.e. that the relation (4) holds.

Given that the relation (9) holds, the relations (8) and (10) are *sufficient* conditions to ensure that the trivial solution $\mathsf{B} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} = \emptyset$ to the relation (3) is avoided. The relations (8) and (10) are also *necessary* to ensure that $\mathsf{B} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \neq \emptyset$ holds in a general context such as the OEM/supplier chain, but in fact not necessary for the case when $\mathbb{E}$ and the elements in $\{\mathbb{E}_i\}_{i=1}^{N}$ are not developed in isolation from each other. An example of such a case is when both the element $\mathbb{E}$ and the elements in $\{\mathbb{E}_i\}_{i=1}^{N}$ are developed within the same company. Due to the fact that the team that develops $\mathbb{E}$ has full access to the elements in $\{\mathbb{E}_i\}_{i=1}^{N}$, the non-trivial solution $\mathsf{B} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \neq \emptyset$ is avoided by composing the elements in a trial-and-error fashion, relying on the expertise of the in-house development teams to make small modifications to the behaviors when needed. Therefore, in order for the relation (4) to hold in such a context, it is sufficient that the respective relations (7) and (9) on the element and the environment hold.

As a conclusion to this section, it is examined whether the element $\mathbb{E}_{lMeter}$ and its environment $\mathbb{E}_{Env_{\mathcal{E}_{LMsys}}(\mathbb{E}_{lMeter})}$ in the context of the set of elements $\mathcal{E}_{LMsys} = \{\mathbb{E}_{bat}, \mathbb{E}_{tank}, \mathbb{E}_{pot}, \mathbb{E}_{lMeter}\}$ shown in Figure 4 are such that the respective conditions (i) and (ii) of Corollary 1 holds with respect to the contract $\mathcal{C}_{lMeter}$ shown in Figure 5. As expressed in the condition (i) of Corollary 1, it must hold that

$$\mathsf{A}_{lMeter} \cap \mathsf{B}_{lMeter} \subseteq \mathsf{G}_{lMeter}, \text{ and} \tag{11}$$

$$\mathsf{A}_{lMeter} \cap \mathsf{G}_{lMeter} \subseteq \mathsf{B}_{lMeter} \ . \tag{12}$$

Furthermore, as expressed in the condition (ii) of Corollary 1, it must hold that

$$\mathsf{B}_{Env_{\mathcal{E}_{LMsys}}(\mathbb{E}_{lMeter})} \subseteq \mathsf{A}_{lMeter}, \text{ and} \tag{13}$$

$$\mathsf{B}_{Env_{\mathcal{E}_{LMsys}}(\mathbb{E}_{lMeter})} \cap \mathsf{G}_{lMeter} \neq \emptyset \ . \tag{14}$$

By applying the operation of intersection, $\mathsf{A}_{lMeter} \cap \mathsf{B}_{Lmeter}$ yields an assertion specified by the equations $f = (v_{branch} - v_{gnd})/5V$ and $l = (v_{branch} -$

$v_{gnd})/5V$. Due to the fact that the equation $f = l$, which specifies G_{lMeter}, can be obtained by combining the equations specifying $\mathsf{A}_{lMeter} \cap \mathsf{B}_{Lmeter}$, the relation (11) holds. Furthermore, since $\mathsf{A}_{lMeter} \cap \mathsf{G}_{lMeter}$ is an assertion specified by the equations $l = f$ and $f = (v_{branch} - v_{gnd})/5V$, which can be combined into the equation $l = (v_{branch} - v_{gnd})/5V$ that specifies B_{lMeter}, the relation (12) also holds.

In accordance with Section 2.3, the behavior of $Env_{\mathscr{A}_{LMsys}}(\mathbb{E}_{lMeter})$ is the intersection of the behaviors of $\mathbb{E}_{tank}$, $\mathbb{E}_{Esys}$, and $\mathbb{E}_{pot}$, which are specified by the equations $f = h$, $h = (v_{branch} - v_{gnd})/(v_{branch} - v_{gnd})$, and $v_{branch} - v_{gnd} = 5V$. Due to the fact that the equation $f = (v_{branch} - v_{gnd})/5V$ can be obtained by combining these equations, the relation (13) holds. Furthermore, since $\mathsf{B}_{Env_{\mathcal{E}_{LMsys}}(\mathbb{E}_{lMeter})}$ does not constrain l, it holds that $\mathsf{B}_{Env_{\mathcal{E}_{LMsys}}(\mathbb{E}_{lMeter})} \cap \mathsf{G}_{lMeter}$ is non-empty. Hence, the relation (14) also holds.

Considering that the relations (11)-(14) hold, Corollary 1 implies that the guarantee G_{lMeter} is non-trivially fulfilled by the composition of $\mathcal{E}_{LMsys}$.

4 Scoping Conditions for Specifying Contracts

Section 3 presented conditions on an element and its environment where the conditions ensure that the guarantee of a given contract is non-trivially fulfilled. Previous general assume-guarantee theories [3, 4, 33, 35–37, 39–41, 43, 44, 47–50, 54–57] that explicitly consider ports, do not allow guarantees and assumptions to be specified over ports that are not in the set of ports of an element. This means that the present paper is strictly more expressive than previous assume-guarantee theories with respect to how a contract can be specified. However, the increased expressiveness is not unlimited since there are necessary restrictions on the set of port variables constrained by the assumptions and the guarantee of a contract in order for an element and its environment to be such that the conditions (i) and (ii) of Corollary 1 hold. Therefore, to facilitate the specification of contracts in practice, this section introduces conditions, called *scoping conditions*, which ensure that these restrictions are not violated without limiting expressiveness.

In order to introduce and motivate these scoping conditions, a definition and two propositions, will first be presented. The definition, which now follows, characterizes a necessary condition in order for conditions (i) and (ii) of Corollary 1 to hold.

Definition 5. An assertion W *restricts* a variable x if W constrains x and there does not exist an assertion W' where $x \notin X_{\mathsf{W}'}$ and $\emptyset \neq \mathsf{W}' \subseteq \mathsf{W}$. □

In accordance with Definition 5, if an assertion W restricts x, then it is necessary for an assertion W' to constrain x in order for W' to non-trivially fulfill W.

As a first example, the guarantee G_{Esys}, expressed through the equation $l = f$, restricts both l and f. Consider an architecture containing an element $\mathbb{E}$ where $\mathsf{B} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})}$ does not constrain both l and f. In accordance with Definition 5, it does not hold that $\emptyset \neq \mathsf{B} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{G}_{Esys}$. Formulated differently, the fact that G_{Esys} restricts a port variable that is not constrained by $\mathsf{B} \cap \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})}$, means both the conditions (i) and (ii) of Corollary 1 will not hold for $\mathbb{E}$ and $Env_{\mathcal{E}}(\mathbb{E})$, respectively.

In the first example, the port variables constrained by G_{Esys} are also the port variables it restricts; however, in general, an assertion can constrain a port variable without restricting it. As a second example, consider an assertion $\mathsf{W}_{x>0\Rightarrow y=0}$ expressed through the logical formula $x > 0 \Rightarrow y = 0$. The assertion $\mathsf{W}_{x>0\Rightarrow y=0}$ constrains y, but it does not restrict it since in the case of e.g. an assertion $\mathsf{W}_{x=0}$ expressed through the equation $x = 0$, it does indeed hold that $\emptyset \neq \mathsf{W}_{x=0} \subseteq \mathsf{W}_{x>0\Rightarrow y=0}$ despite the fact that $\mathsf{W}_{x=0}$ does not constrain y.

As shown in the second example, under the conditions that an assertion W constrains a variable x, but does not restrict it, then it is possible that there exists a case where another assertion W'' does not constrain x, but where it still holds that $\emptyset \neq \mathsf{W}'' \subseteq \mathsf{W}$. As will be shown in the following proposition, under such conditions, it holds that there exists another assertion W' that does not constrain x, but where $\emptyset \neq \mathsf{W}'' \subseteq \mathsf{W}'$ holds regardless of the specific runs that are in W''. This means that if the intent is that W is to specify an assertion that is to non-trivially fulfill W and not constrain x, then it is possible to replace W with the assertion W' that does not constrain x, but that still specifies the exact same assertions.

Proposition 4. *Given a set of variables X and an assertion W, there exists an assertion W' where $X_{\mathsf{W}'} \subseteq X$ such that for each assertion W'' where $X_{\mathsf{W}''} \subseteq X$, it holds that*

$$\emptyset \neq \mathsf{W}'' \subseteq \mathsf{W}' \text{ if and only if } \emptyset \neq \mathsf{W}'' \subseteq \mathsf{W} .$$

Lemma 1. *Given two assertions W and W' where $\mathsf{W} \cup \mathsf{W}' \neq \emptyset$, it holds that $X_{\mathsf{W} \cup \mathsf{W}'} \subseteq X_{\mathsf{W}} \cup X_{\mathsf{W}'}$.*

The proof of Lemma 1 is found in Appendix I. The proof of Proposition 4 follows.

Proof. Assume that W' is the union of each assertion W'' where $X_{\mathsf{W}''} \subseteq X$ and $\emptyset \neq \mathsf{W}'' \subseteq \mathsf{W}$. The rest of the proof trivially follows from Lemma 1. □

Note that in order for there to exist an assertion W'' where $X_{\mathsf{W}''} \subseteq X$ and $\emptyset \neq \mathsf{W}'' \subseteq \mathsf{W}$, in accordance with Definition 5, it is necessary that W does not restrict variables that are not in X.

Given the previously presented assertion $\mathsf{W}_{x>0\Rightarrow y=0}$ and the set $\{x\}$. In accordance with Proposition 4, there exists an assertion W' where $X_{\mathsf{W}'} \subseteq \{x\}$ such that for each assertion W'' where $X_{\mathsf{W}''} \subseteq \{x\}$, it holds that $\emptyset \neq \mathsf{W}'' \subseteq \mathsf{W}_{x>0\Rightarrow y=0}$ if and only if $\emptyset \neq \mathsf{W}'' \subseteq \mathsf{W}'$. For example, W' can be expressed through the inequality $x \leq 0$.

Proposition 5. *Given two assertions* W *and* W' *where* $\mathsf{W} \cap \mathsf{W}' \neq \emptyset$*, it holds that* $X_{\mathsf{W}\cap\mathsf{W}'} \subseteq X_{\mathsf{W}} \cup X_{\mathsf{W}'}$.

The proof of Proposition 5 is found in Appendix I.

Definition 5 and Propositions 4 and 5 will now be applied on two examples to motivate the need and the basis for the scoping rules that will be proposed for specifying contracts. In the following two examples, the considered use case is to establish that the guarantee of a contract is non-trivially fulfilled in a set of elements containing an element with a set of port variables X_{Esys} and where X_{tank} is the set of port variables of the environment of this element.

What will be shown in the two examples is that, regardless of the specific runs in the assumptions and guarantee of a contract, it is indeed necessary that the assumptions and guarantee do not restrict variables in $X_{Esys} \cup X_{tank}$ in order for the condition (i) and (ii) of Corollary 1 to hold. Furthermore, it will also be shown that if the assumptions and guarantee constrain port variables in $X_{Esys} \cup X_{tank}$ without restricting them, then the assumptions and guarantee can be reformulated to constrain a subset of $X_{Esys} \cup X_{tank}$ and still specify the exact same element and environment behaviors constraining a subset of X_{Esys} and X_{tank}, respectively.

Example 1a. Consider Figure 7a that shows a contract

$$\mathcal{C}'_{Esys} = (\{\mathsf{A}'_{Esys}\}, \mathsf{G}'_{Esys}, X_{Esys}) \ .$$

Assume that the conditions (i) and (ii) of Corollary 1 hold respectively for $\mathbb{E}_{Esys}$ and $\mathbb{E}_{tank}$ with respect to $\mathcal{C}'_{Esys}$. From the condition (ii), it follows that $\emptyset \neq \mathsf{B}_{tank} \subseteq \mathsf{A}'_{Esys}$. As shown in Figure 7a, the port variable v_{gnd} is constrained by A'_{Esys}, but v_{gnd} is not a port variable of $\mathbb{E}_{tank}$, i.e. the environment of $\mathbb{E}_{Esys}$. In accordance with Definition 2, this means that v_{gnd} is not constrained by B_{tank}. This means, in accordance with Definition 5 and considering that $\emptyset \neq \mathsf{B}_{tank} \subseteq \mathsf{A}'_{Esys}$, that A'_{Esys} does not restrict v_{gnd}. That is, the fact that the condition (ii) holds, implies that A'_{Esys} does not restrict v_{gnd}.

Furthermore, considering X_{tank} as given, in accordance with Proposition 4, there exists an assertion A^{new}_{Esys} that constrains a subset of X_{tank} and where, for each element $(X_{tank}, \mathsf{B}'_{tank})$, it holds that $\emptyset \neq \mathsf{B}'_{tank} \subseteq \mathsf{A}'_{Esys}$ if and only if $\emptyset \neq \mathsf{B}'_{tank} \subseteq \mathsf{A}^{new}_{Esys}$.

This example shows that it is necessary that the assumption $\mathsf{A}_{\mathscr{A}}$ does not restrict any port variable that is not in $X_{Env_{\mathcal{E}}(\mathbb{E})}$. Furthermore, if $\mathsf{A}_{\mathscr{A}}$ does constrain a port variable $x \notin X_{Env_{\mathcal{E}}(\mathbb{E})}$ without restricting it, then it is possible to replace A'_{Esys} with an assertion A^{new}_{Esys} that does not constrain x, but that still specifies the exact same behaviors constraining a subset of X_{tank}. □

Example 1b. In Figure 7a, a contract $\mathcal{C}'_{Esys} = (\{\mathsf{A}'_{Esys}\}, \mathsf{G}'_{Esys}, X_{Esys})$ is shown.

Similar to Example 1a, assume that the conditions (i) and (ii) of Corollary 1 hold respectively for $\mathbb{E}_{Esys}$ and $\mathbb{E}_{tank}$ with respect to $\mathcal{C}''_{Esys}$. The conditions (i) and (ii) imply that $\emptyset \neq \mathsf{A}''_{Esys} \cap \mathsf{B}_{Esys} \subseteq \mathsf{G}''_{Esys}$. As shown in Figure 7b, the port variable v_{gnd} is constrained by G''_{Esys}, but v_{gnd} is not a port variable of $\mathbb{E}_{Esys}$. This and considering that $v_{gnd} \notin X_{\mathsf{A}''_{Esys}}$ imply, in accordance with Definition 2 and Proposition 5, that v_{gnd} is not constrained by $\mathsf{A}''_{Esys} \cap \mathsf{B}_{Esys}$. It follows, in accordance with Definition 5 and considering that $\emptyset \neq \mathsf{A}''_{Esys} \cap \mathsf{B}_{Esys} \subseteq \mathsf{G}''_{Esys}$, that G''_{Esys} does not restrict v_{gnd}. That is, the fact that the condition (i) and (ii) hold, implies that G''_{Esys} does not restrict v_{gnd}.

In addition, considering the set $X_{\mathsf{A}''_{Esys}} \cup X_{Esys}$ as given, in accordance with Propositions 4 and 5, there exists an assertion G^{new}_{Esys} that constrains a subset of $X_{\mathsf{A}''_{Esys}} \cup X_{Esys}$ and where, for each element $(X_{Esys}, \mathsf{B}'_{Esys})$, it holds that $\emptyset \neq \mathsf{A}''_{Esys} \cap \mathsf{B}'_{Esys} \subseteq \mathsf{G}''_{Esys}$ if and only if $\emptyset \neq \mathsf{A}''_{Esys} \cap \mathsf{B}'_{Esys} \subseteq \mathsf{G}^{new}_{Esys}$. That is, similar to Example 1a, it is possible to replace G''_{Esys} with an assertion G^{new}_{Esys} that does not constrain v_{gnd}, but that still specifies the

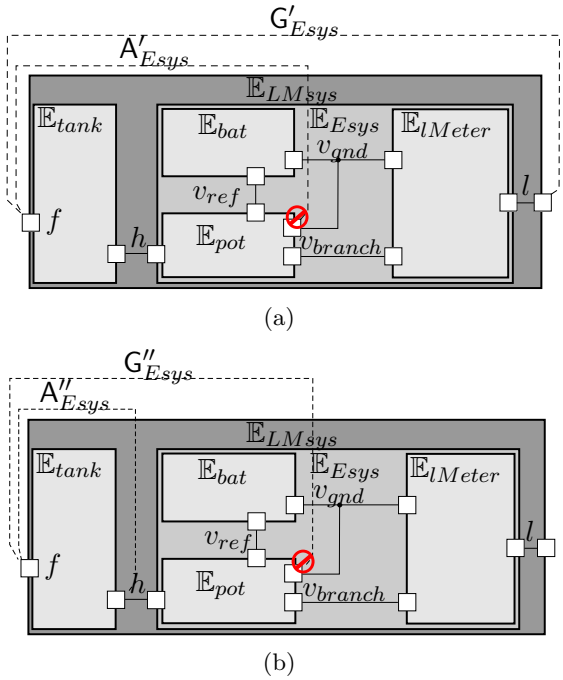


Figure 7. In a) and b), two respective contracts $(\{\mathsf{A}'_{Esys}\}, \mathsf{G}'_{Esys}, X_{Esys})$ and $(\{\mathsf{A}''_{Esys}\}, \mathsf{G}''_{Esys}, X_{Esys})$ are shown where none of them are scope-compliant with respect to X_{tank}.

exact same behaviors constraining a subset of $X_{\mathsf{A}''_{Esys}} \cup X_{Esys}$. □

Consider a contract $(\mathscr{A}, \mathsf{G}, X)$ and a set of port variables $X_{Env_{\mathcal{E}}(\mathbb{E})}$. As expressed in Example 1a, in order for the condition (ii) of Corollary 1 to hold for an element $(X_{Env_{\mathcal{E}}(\mathbb{E})}, \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})})$, it is necessary that the assumption $\mathsf{A}_{\mathscr{A}}$ does not restrict any port variable that is not in $X_{Env_{\mathcal{E}}(\mathbb{E})}$. Similarly, as expressed in Example 1b, in order for the condition (i) of Corollary 1 to hold for an element (X, B), it is necessary that the guarantee G does not restrict any port variable that is not in $X \cup X_{\mathsf{A}_{\mathscr{A}}}$.

Furthermore, consider that $\mathsf{A}_{\mathscr{A}}$ and G constrain port variables that are not in the respective sets $X_{Env_{\mathcal{E}}(\mathbb{E})}$ and $X \cup X_{\mathsf{A}_{\mathscr{A}}}$, but where $\mathsf{A}_{\mathscr{A}}$ and G do not restrict these port variables. As indicated in Examples 1a and 1b, in such a case, it is, in fact, redundant to constrain such port variables. That is, the examples indicate that $\mathsf{A}_{\mathscr{A}}$ and G could be reformulated to *not constrain such port variables* and yet *specify the same element and environment behavior.*

These indications are formalized in the following theorem.

Theorem 2. *Consider a contract* $(\mathscr{A}, \mathsf{G}, X)$ *and set of variables* $X_{Env_{\mathcal{E}}(\mathbb{E})}$. *It holds that there exists a contract* $(\mathscr{A}', \mathsf{G}', X)$ *where:*

a) $X_{\mathsf{A}_{\mathscr{A}'}} \subseteq X_{Env_{\mathcal{E}}(\mathbb{E})}$; *and*

b) $X_{\mathsf{G}'} \subseteq X_{Env_{\mathcal{E}}(\mathbb{E})} \cup X$

such that for each set of elements containing an element (X, B) *and where the pair* $(X_{Env_{\mathcal{E}}(\mathbb{E})}, \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})})$ *is the environment of* (X, B), *it holds that:*

i') $\mathsf{A}_{\mathscr{A}'} \cap \mathsf{B} \subseteq \mathsf{G}'$ *and* $\mathsf{A}_{\mathscr{A}'} \cap \mathsf{G}' \subseteq \mathsf{B}$, *and*

ii') $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}'}$ *and* $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G}' \neq \emptyset$,

if and only if

i) $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ *and* $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$, *and*

ii) $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ *and* $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$.

The proof of Theorem 2 is found in Appendix I.

Consider the task of specifying a contract $(\mathscr{A}, \mathsf{G}, X)$ with the intent that the condition (i) and (ii) are to hold respectively for an element (X, B) and its environment $(X_{Env_{\mathcal{E}}(\mathbb{E})}, \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})})$ in a set of elements. As expressed in Theorem 2, for such a task, it is sufficient to only allow contracts to be specified in accordance with the conditions (a) and (b); that is, no expressiveness is lost by only allowing such contracts. In the following, given a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and a set of variables $X_{Env_{\mathcal{E}}(\mathbb{E})}$, the conditions (a) and (b) will be called *scoping conditions for* $\mathcal{C}$ *and* $X_{Env_{\mathcal{E}}(\mathbb{E})}$.

Consider that either or both of the scoping conditions (a) and (b) are violated for a given contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and a set of variables $X_{Env_{\mathcal{E}}(\mathbb{E})}$. For example, consider that $X_{\mathsf{A}_{\mathscr{A}'}} \setminus X_{Env_{\mathcal{E}}(\mathbb{E})} = \{x\}$. Notably, either $\mathsf{A}_{\mathscr{A}'}$ restricts x or it constrains it without restricting it. As previously mentioned and in accordance with Definition 5, the former case ensures that the conditions (i) and (ii) cannot hold. Considering the latter case, in accordance with Theorem 2, constraining x is indeed redundant. Thus, the scoping conditions of Theorem 2 constitute as checks that either detect the violation of the conditions (i) and (ii) of Corollary 1 or redundantly constrained port variables.

Note that in accordance with Section 2.1.2, the set of variables constrained by an assertion can only be derived from the runs that the assertion contains. However, as indicated in Section 2.1, in practice, an assertion would typically be expressed through a constraint, explicitly specified over a set of variables, e.g. as the assertion $\mathsf{W}_{y=x}$ shown in Figure 2a. Assuming the generic case when the set of variables over which the constraint is specified is equal to the set of variables constrained by the assertion, the fact that the scoping conditions (a) and (b) hold for a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and a set of port variables $X_{Env_{\mathcal{E}}(\mathbb{E})}$, can actually be established by considering only X, $X_{Env_{\mathcal{E}}(\mathbb{E})}$, and the sets of port variables over which the assumptions $\mathsf{A}_{\mathscr{A}}$ and the guarantee G are specified.

To illustrate the use of the conditions (a) and (b) of Theorem 2, consider again Examples 1a and 1b shown in Figure 7. The scoping condition (a) is violated for the contract $\mathcal{C}'_{Esys}$ and X_{tank}, due to the fact that $X_{\mathsf{A}_{Esys}} \not\subseteq X_{tank}$. Furthermore, considering the contract $\mathcal{C}''_{Esys}$ and the set of port variables X_{tank}, the scoping condition (b) does not hold since $X_{\mathsf{G}''_{Esys}} \not\subseteq X_{tank} \cup X_{Esys}$. Thus, checking whether the scoping conditions (a) and (b) hold or not, can be done without explicitly considering the runs that are in these assumptions and guarantees. As previously mentioned, the violation of the scoping conditions means that either the conditions (i) and (ii) of Corollary 1 are violated or that more port variables are constrained than what is necessary.

5 Contract Properties Consistency and Compatibility

Section 4 described necessary restrictions on the sets of port variables constrained by assumptions and the guarantee of a contract, in order for the conditions (i) and (ii) of Corollary 1 to hold. In contrast to Section 4, this section presents sufficient and necessary conditions for the *existence of a set of elements* where an element and its environment is such that the conditions (i) and (ii) of Corollary 1 holds with respect to the contract. If such a set of elements exists, then the contract is said to be *consistent* and *compatible*.

In order to get a better understanding of when the properties consistency and compatibility are relevant, the scenario presented in Section 3.2 is examined. Considering the phase (I) of the scenario, the expectation of the OEM when handing over the contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ to the supplier, is that the supplier will deliver an element in the phase (II) such that the

condition (i) of Corollary 1 holds with respect to $\mathcal{C}$. However, in order for the supplier to be able to meet this expectation from the OEM, the contract $\mathcal{C}$ needs to be such that there actually exists an element (X, B) that is such that the condition (i) of Corollary 1 holds. If such an element exists, the contract $\mathcal{C}$ will be referred to as a *consistent* contract.

Furthermore, in the phase (III), the OEM also has the intent of integrating the element $\mathbb{E}$ delivered by the supplier with a set of elements $\{\mathbb{E}_i\}_{i=1}^N$ such that the composition of $\{\mathbb{E}\} \cup \{\mathbb{E}_i\}_{i=1}^N$ non-trivially fulfills the guarantee G. However, in order for this to be possible, there needs to exists at least one set of elements containing $\mathbb{E}$ where the environment of the element is such that the condition (ii) of Corollary 1 holds with respect to $\mathcal{C}$. If such a set of elements exists, then the contract $\mathcal{C}$ will be referred to as a *compatible* contract.

Now that the concepts of consistency and compatibility have been introduced in the context of a scenario, formal definitions follow. Considering that the definitions quantify over elements and set of elements, complementary sufficient and necessary conditions that can be established to hold on the contract alone, will also be presented.

Definition 6 (Consistent Contract)**.** A contract $(\mathscr{A}, \mathsf{G}, X)$ is *consistent* if there exists an element $\mathbb{E} = (X, \mathsf{B})$ such that

a) $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$, and

b) $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$. □

Definition 6 corresponds to an instantiation of the abstract definition of consistency in the meta theory of contracts in [31] using the condition (i) of Corollary 1. Definition 6 is also closely related to definitions of consistency and compatibility in [3, 5], and to the definition of *realizability* in [70], but where Definition 6, in contrast to the definitions in [3, 5, 70], considers a context where the contract $(\mathscr{A}, \mathsf{G}, X)$ is not necessarily limited to set of port variables X and where X does not need to be partitioned into inputs and outputs.

A sufficient and necessary condition of Definition 6 now follows.

Theorem 3. *A contract* $(\mathscr{A}, \mathsf{G}, X)$ *is* consistent *if and only if*

$$\mathsf{A}_{\mathscr{A}} \cap \widehat{proj}_X(\mathsf{A}_{\mathscr{A}} \cap \mathsf{G}) \subseteq \mathsf{G} \ .$$

Lemma 2. *Given two assertions* W *and* W'*, and a set of variables* X*, it holds that* $\widehat{proj}_X(\mathsf{W}) \subseteq \widehat{proj}_X(\mathsf{W}')$*, if* $\mathsf{W} \subseteq \mathsf{W}'$*.*

The proof of Lemma 2 is found in Appendix I. The proof of Theorem 3 now follows.

Proof. Consider a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$.

For the only-if part, assume that $\mathcal{C}$ is consistent. In accordance with Definition 6, this means that there exists an element $\mathbb{E} = (X, \mathsf{B})$ such that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ and $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$. In accordance with Lemma 2, this means that it holds that $\widehat{proj}_X(\mathsf{A}_{\mathscr{A}} \cap \mathsf{G}) \subseteq \widehat{proj}_X(\mathsf{B})$. This and the fact that $\widehat{proj}_X(\mathsf{B}) = \mathsf{B}$ in accordance with Definition 2 and Proposition 1, it follows that $\widehat{proj}_X(\mathsf{A}_{\mathscr{A}} \cap \mathsf{G}) \subseteq \mathsf{B}$. This and the relation $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ imply that $\mathsf{A}_{\mathscr{A}} \cap \widehat{proj}_X(\mathsf{A}_{\mathscr{A}} \cap \mathsf{G}) \subseteq \mathsf{G}$.

For the if part, assume that the relation $\mathsf{A}_{\mathscr{A}} \cap \widehat{proj}_X(\mathsf{A}_{\mathscr{A}} \cap \mathsf{G}) \subseteq \mathsf{G}$ holds. Assume that $\mathbb{E} = (X, \mathsf{B})$ is an element where $\mathsf{B} = \widehat{proj}_X(\mathsf{A}_{\mathscr{A}} \cap \mathsf{G})$, which means that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$. In accordance with the relations (1) and (2), it holds that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \widehat{proj}_X(\mathsf{A}_{\mathscr{A}} \cap \mathsf{G})$. This and considering that it holds that $\mathsf{B} = \widehat{proj}_X(\mathsf{A}_{\mathscr{A}} \cap \mathsf{G})$ imply that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$, which concludes the proof. □

In contrast to Definition 6, Theorem 3 supports a way of establishing whether a contract is consistent or not without the need for iterating through each element with a set of port variables X in order to determine if there exists an element $\mathbb{E} = (X, \mathsf{B})$ that is such that the condition (i) of Corollary 1 holds with respect to $\mathcal{C}$.

Definition 7 (Compatible Contract)**.** A contract $(\mathscr{A}, \mathsf{G}, X)$ is *compatible* if there exists an element $\mathbb{E} = (X, \mathsf{B})$ and a set of elements $\mathcal{E}$ containing $\mathbb{E} = (X, \mathsf{B})$, such that

a) $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$, and

b) $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$. □

Definition 7 corresponds to an instantiation of the abstract definition of compatibility in the meta theory of contracts in [31] using the condition (ii) of Corollary 1. Definition 7 is also closely related to the definitions of compatibility in [3, 5], but where Definition 7, in contrast to the definitions

in [3, 5], considers a context where the contract $(\mathscr{A}, \mathsf{G}, X)$ is not necessarily limited to set of port variables X and where X does not need to be partitioned into inputs and outputs.

A sufficient and necessary condition of compatibility now follows.

Theorem 4. *A contract* $(\mathscr{A}, \mathsf{G}, X)$ *is* compatible *if and only if*

$$\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \neq \emptyset \ .$$

Proof. Consider a contract $(\mathscr{A}, \mathsf{G}, X)$.

For the if part, assume that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \neq \emptyset$. This implies that there exists at least one run ω in $\mathsf{A}_{\mathscr{A}}$ that is also in $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G}$. Assume that $\mathcal{E}_0$ is a set of elements containing an element $\mathbb{E} = (X, \mathsf{B})$ such that $\mathsf{B}_{Env_{\mathcal{E}_0}(\mathbb{E})} = \{\omega\}$. This implies that there exists a set of elements containing an element $\mathbb{E} = (X, \mathsf{B})$, such that the relations (i) and (ii) of Definition 7 hold.

For the only-if part, assume that $(\mathscr{A}, \mathsf{G}, X)$ is compatible. In accordance with Definition 7, this means that there exists a set of elements $\mathcal{E}$ containing an element $\mathbb{E} = (X, \mathsf{B})$, such that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$. It trivially follows that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \neq \emptyset$, which completes the proof. □

In contrast to Definition 7, Theorem 4 supports a way of establishing whether a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G})$ is compatible or not, without the need for iterating through each set of elements containing an element with a set of port variables X in order to determine if there exists a set of elements where the environment of an element $\mathbb{E} = (X, \mathsf{B})$ is such that the condition (ii) of Corollary 1 holds.

As a conclusion to this section, a scenario is examined where a supplier wants to outsource the development of a level meter by the use of the contract $\mathcal{C}_{lMeter}$ shown in Figure 5. In order for the supplier and the client to complete the phases (I-III) in the scenario described in Section 3.2 such that the relation (4) holds, the contract $\mathcal{C}_{lMeter}$ must be consistent and compatible.

For this to be the case, Theorem 3 and Theorem 4 express that it is necessary and sufficient that the following holds:

$$\mathsf{A}_{lMeter} \cap \mathsf{G}_{lMeter} \neq \emptyset; \text{ and} \tag{15}$$

$$\mathsf{A}_{lMeter} \cap \widehat{proj}_{X_{lMeter}}(\mathsf{A}_{lMeter} \cap \mathsf{G}_{lMeter}) \subseteq \mathsf{G}_{lMeter} \ . \tag{16}$$

Considering that $\mathsf{A}_{lMeter} \cap \mathsf{G}_{lMeter}$ is an assertion specified by the equations $f = \frac{v_{branch}-v_{gnd}}{5}$ and $l = f$, which obviously have intersecting solutions, the relation (15) holds. The extended projection of this assertion onto $X_{lMeter} = \{l, v_{branch}, v_{gnd}\}$, i.e. the assertion $\widehat{proj}_{X_{lMeter}}(\mathsf{A}_{lMeter} \cap \mathsf{G}_{lMeter})$, is specified by $l = \frac{v_{branch}-v_{gnd}}{5}$. The intersection of A_{lMeter} and the assertion $\widehat{proj}_{X}(\mathsf{A}_{lMeter} \cap \mathsf{G}_{lMeter})$ yields an assertion specified by the equations $f = \frac{v_{branch}-v_{gnd}}{5}$ and $l = \frac{v_{branch}-v_{gnd}}{5}$, which can be combined into the equation $l = f$. This means that the relation (16) holds. Therefore, it can be concluded that the contract $\mathcal{C}_{lMeter}$ is consistent and compatible and is, thus, an appropriate specification to be used for outsourced development.

6 Hierarchical Structuring of Contracts

Consider a pair $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$, characterizing a *two-level contract hierarchy* such that $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ is a contract at the first level and $\{\mathcal{C}_i\}_{i=1}^N$ is a set of contracts where $\mathcal{C}_i = (\mathscr{A}_i, \mathsf{G}_i, X_i)$ and $X \subseteq \bigcup_{i=1}^N X_i$, at the second level. This section establishes the following property of the two level contract hierarchy $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$: for each set of elements $\{(X_i, \mathsf{B}_i)\}_{i=1}^N$ where each element (X_i, B_i) is such that the condition (i) of Corollary 1 holds with respect to $(\mathscr{A}_i, \mathsf{G}_i, X_i)$, it holds that the composition (X, B) of $\{(X_i, \mathsf{B}_i)\}_{i=1}^N$ onto X is such that the condition (i) of Corollary 1 holds with respect to $\mathcal{C}$. If this property holds, then the two-level contract hierarchy $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$ is said to be *proper*.

Definition 8 (Proper Contract Hierarchy)**.** Given a set of contracts $\{\mathcal{C}_i = (\mathscr{A}_i, \mathsf{G}_i, X_i)\}_{i=1}^N$ and a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ where $X \subseteq \bigcup_{i=1}^N X_i$, *the two-level contract hierarchy* $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$ *is proper* if, for each set of elements $\{(X_i, \mathsf{B}_i)\}_{i=1}^N$, it holds that

$$(\forall i : \mathsf{A}_{\mathscr{A}_i} \cap \mathsf{B}_i \subseteq \mathsf{G}_i \text{ and } \mathsf{A}_{\mathscr{A}_i} \cap \mathsf{G}_i \subseteq \mathsf{B}_i) \Longrightarrow \mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G} \text{ and } \mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B},$$

where (X, B) is the composition of $\{(X_i, \mathsf{B}_i)\}_{i=1}^N$ onto X. □

Definition 8 is in accordance with the general principle of *compositionality* [29, 30] since the fact that the element (X, B) is such that the relations $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ and $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$ hold can be inferred from establishing that each element (X_i, B_i) is such that the relations $\mathsf{A}_{\mathscr{A}_i} \cap \mathsf{B}_i \subseteq \mathsf{G}_i$ and $\mathsf{A}_{\mathscr{A}_i} \cap \mathsf{G}_i \subseteq \mathsf{B}_i$ hold. The compositional approach of indirectly establishing that (X, B) is

such that the condition (i) of Corollary 1 holds with respect to $\mathcal{C}$, is needed when a direct approach is not feasible due to e.g. the complexity of (X, B).

Despite considering dissimilar conditions than those presented in Corollary 1 in the present paper, Definition 8 corresponds, in essence, to the definitions of *dominance* in [37, 71] where [71] offers a minor extension to the definitions in [33, 42]. However, in contrast to [71] where ports are not considered and to [37] where guarantees must be limited to the set of ports of a component, Definition 8 considers a context where port variables are explicit, but where the contracts are not necessarily limited to the sets of port variables of elements.

In order to provide further understanding of when Definition 8 is relevant, a scenario is presented. The scenario is in the context of an OEM/supplier chain as described in Section 3, but the principles are equally valid for any design context where clear separations of responsibilities are desired, also within a single company. Specifically, the scenario consists of three phases:

I') the OEM establishes a two-level contract hierarchy $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$ where $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and where each contract $\mathcal{C}_i = (\mathscr{A}_i, \mathsf{G}_i, X_i)$ is handed from the OEM to a supplier and where $X \subseteq \bigcup_{i=1}^N X_i$;

II') each supplier develops an element $\mathbb{E}_i = (X_i, \mathsf{B}_i)$ such that the condition (i) of Corollary 1 holds with respect to $\mathcal{C}_i$; and

III') the OEM integrates the set of elements $\{\mathbb{E}_i\}_{i=1}^N$ into an element $\mathbb{E} = (X, \mathsf{B})$ that is the composition of $\{\mathbb{E}_i\}_{i=1}^N$ onto X and is such that the condition (i) of Corollary 1 holds with respect to $\mathcal{C}$.

In order to enable an integration of the elements $\{\mathbb{E}_i\}_{i=1}^N$ into the element $\mathbb{E}$ in the phase (III'), the fact that the element $\mathbb{E}$ is such that the condition (i) of Corollary 1 holds with respect to $\mathcal{C}$ needs to follow from the completion of the phase (II'). In order for this to be the case, in the phase (I'), the two-level contract hierarchy $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$ needs to proper.

In order to find a two-level contract hierarchy $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$ that is proper, a graph, called a *composition structure*, is introduced in Section 6.1. Based on a composition structure, a theorem that expresses sufficient conditions for a contract hierarchy to be proper is presented in Section 6.2. Despite considering dissimilar formalisms than the present paper, the sufficient conditions that will be presented for a contract hierarchy to be proper, corresponds, in essence, to the *compositional proof step* [72] in assume-guarantee

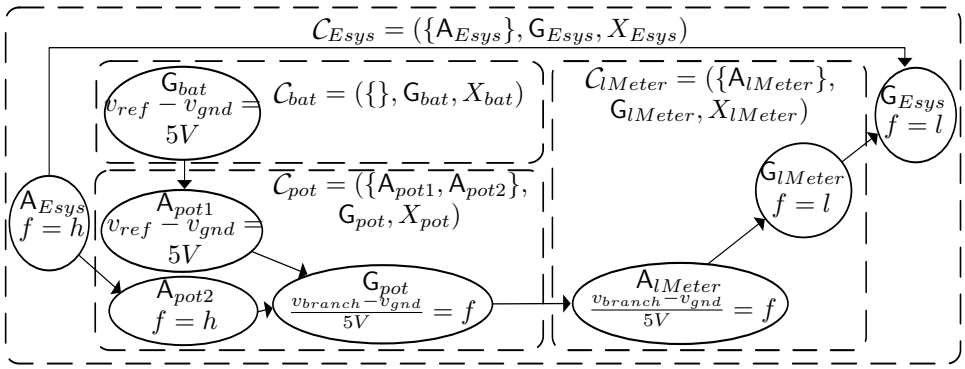


Figure 8. A composition structure of two-level contract hierarchy $(\mathcal{C}_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\})$.

theories such as [32, 45, 49], and also to sufficient conditions of dominance in [37]. However, in contrast to [32, 37, 45, 49], the sufficient conditions in Section 6.2 are based on the concept of a graph, or more specifically, a composition structure.

6.1 Composition Structures of Contract Hierarchies

Prior to presenting the formal definition of a composition structure, the concept is introduced in an informal manner by structuring a two-level contract hierarchy $(\mathcal{C}_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\})$. The contracts express specifications for the elements of the electric-system of an LM-system, e.g. the one shown in Figure 4.

Consider the assumptions and the guarantee of each contract in the set of contracts $\{\mathcal{C}_{Esys}, \mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\}$ being structured as nodes in a directed graph, as shown in Figure 8 where the boxes with rounded corners and dashed edges have been added to also show the two-level contract hierarchy. The set of incoming arcs to a guarantee G from a set of assumptions $\mathscr{A}$, represents that $\mathscr{A}$ and G are in the same contract, e.g. the arc from A_{Esys} to the guarantee G_{Esys} represents the contract $(\{\mathsf{A}_{Esys}\}, \mathsf{G}_{Esys}, X_{Esys})$.

The set of incoming arcs to an assumption A from a set of assertions $\{\mathsf{W}_i\}_{i=1}^N$ where W_i is either an assumption or a guarantee, represents the intent $\bigcap_{i=1}^N \mathsf{W}_i \subseteq \mathsf{A}$. For example, the arc to A_{pot2} from the assumption A_{Esys}, represents the intent of $\mathsf{A}_{Esys} \subseteq \mathsf{A}_{pot2}$.

The set of incoming arcs to a guarantee G from a set of guarantees

$\{\mathsf{G}_j\}_{j=1}^{M}$ represents the intent of $\bigcap_{j=1}^{M} \mathsf{G}_j \subseteq \mathsf{G}$. For example, the arc from the guarantee G_{lMeter} to G_{Esys}, represents the intent of $\mathsf{G}_{lMeter} \subseteq \mathsf{G}_{Esys}$.

Now that the concept of a composition structure has been introduced informally, the formal definition follows.

Definition 9 (Composition Structure of Contract Hierarchy)**.** Given a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and a set of contracts $\{\mathcal{C}_i = (\mathscr{A}_i, \mathsf{G}_i, X_i)\}_{i=1}^{N}$ where $X \subseteq \bigcup_{i=1}^{N} X_i$, a *composition structure* $\mathfrak{D}$ *of two-level contract hierarchy* $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^{N})$ is a Directed Acyclic Graph (DAG), such that:

a) the guarantees G_i, the assumptions in each $\mathscr{A}_i$, the assumptions in $\mathscr{A}$, and the guarantee G are the nodes in $\mathfrak{D}$;

b) G has no successors;

c) each assumption in $\mathscr{A}$ has no predecessor;

d) at least one G_i is a direct predecessor of G;

e) each assumption in each $\mathscr{A}_i$ has at least one predecessor;

f) the assumptions in $\mathscr{A}_i$ are the only direct predecessors of each G_i;

g) G_i is the only direct successor of each assumption in each $\mathscr{A}_i$; and

h) G is a direct successor of each assumption in $\mathscr{A}$. □

As described in the beginning of this section and in Definition 9, a composition structure represents a structuring of a two-level contract hierarchy as expressed in the condition (a) where: the set of incoming arcs to a guarantee from a set of assumptions represents that the assumptions and the guarantee are in the same contract as expressed in the conditions (f) and (h); the intent is that at least one guarantee G_i or an assumption in $\mathscr{A}$ should be a subset of each assumption in $\mathscr{A}_i$, as expressed in the conditions (b) and (g), and the condition (e) in particular; and the intent is that at least one guarantee G_i should be a subset of the guarantee G, as expressed in the condition (d). The condition (c), and further also the conditions (b), (f), and (g), disallow the existence of any other arcs from those not already mentioned above.

Out of the general assume-guarantee theories [3, 4, 31–57], only [46] considers a graph-based approach for structuring contracts. However, in

contrast to [46] where the aim of the structuring is to be able to verify a set of contracts with *circular dependencies* (see Remark 1 in the end of this section), a composition structure represents a structuring of a two-level contract hierarchy in general.

Apart from general assume-guarantee theories, composition structures do have a lot in common with Goal Oriented Requirements Engineering (GORE) models, see e.g. I* [73] or KAOS [74] or [75] for a survey, where [73, 74] draw on ideas presented in [76–78]. The main difference is, again, that while a composition structure represents a structuring of a two-level contract hierarchy in general, GORE models are more specific since the use of assumptions, also called *expectations*, in GORE models are strictly limited to top-level specifications that split the responsibilities between a *SW system* and its environment. Furthermore, a similar concept to a contract structure is presented in [79, 80] based on Bayesian networks, but with the specific focus to model failure propagation.

Given a composition structure $\mathfrak{D}$ of a contract hierarchy $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$, in accordance with Definition 9, each assumption and each guarantee in $\mathcal{C}$ and $\mathcal{C}_i$ are nodes in $\mathfrak{D}$. There are, however, cases when re-using an assumption or a guarantee would be preferred [15], e.g. if two guarantees rely on the same assumption or if a guarantee is equal to an assumption. In practice, such a case can be represented by either the use of a single node or to label one node as a copy of another.

Remark 1 (Circular Reasoning). Since a composition structure is a directed *acyclic* graph where the assumptions and guarantees are the nodes, 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 [32] or the timing model [46]. See also [81, 82] for more discussions on such matters. □

6.2 Sufficient Conditions for Proper Contract Hierarchy

This section presents sufficient conditions for a two-level contract hierarchy $(\mathcal{C} = (\mathscr{A}, \mathsf{G}, X), \{\mathcal{C}_i = (\{\mathsf{A}_{ij}\}_{j=1}^{M_i}, \mathsf{G}_i, X_i)\}_{i=1}^N)$ to be proper in accordance with Definition 8, based on the concept of a composition structure. The sufficient conditions supports a way of establishing that $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$ is proper, without having to iterate through each possible set of elements $\{(X_i, \mathsf{B}_i)\}_{i=1}^N$ to determine that their composition onto X is such that the condition (i)

of Corollary 1 holds with respect to $\mathcal{C}$, if each element (X_i, B_i) is such that the condition (i) of Corollary 1 holds with respect to $(\mathscr{A}_i, \mathsf{G}_i, X_i)$.

Consider a composition structure of $\mathfrak{D}$ of $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$. Let $dPred()$ denote a function that takes a node W in $\mathfrak{D}$ as input and returns the set of all nodes that are direct predecessors of W. As presented in Section 6.1, the composition structure $\mathfrak{D}$ represents the intent that

$$\bigcap\nolimits_{\mathsf{W}\in dPred(\mathsf{G})\cap\{\mathsf{G}_i\}_{i=1}^N} \mathsf{W} \subseteq \mathsf{G}, \text{ and} \tag{17}$$

$$\bigcap\nolimits_{\mathsf{W}\in dPred(\mathsf{A}_{ij})} \mathsf{W} \subseteq A_{ij}, \text{ for each } i, j. \tag{18}$$

As an example, the composition structure of $(\mathcal{C}_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\})$ in Figure 8 represents the intent that the relation $\mathsf{G}_{lMeter} \subseteq \mathsf{G}_{Esys}$ holds, and furthermore that the relations $\mathsf{G}_{bat} \subseteq \mathsf{A}_{pot1}$, $\mathsf{A}_{Esys} \subseteq \mathsf{A}_{pot2}$, $\mathsf{G}_{pot} \subseteq \mathsf{A}_{lMeter}$ hold.

However, as will be shown in the following illustrative Examples 2a, 2b, and 2c, the fact that the relations (17) and (18) hold for $\mathfrak{D}$ does not imply that $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$ is proper. As a quick overview, Examples 2a and 2b will show that properness cannot be ensured if either the assumptions $\mathsf{A}_{\mathscr{A}}$ or the Guarantee G of the contract $\mathcal{C}$ constrains port variables in the set $\bigcup_{i=1}^N X_i \setminus X$. Example 2c shows the need to introduce an additional condition for the specific purpose of ensuring that the relation $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$ (of the condition (i) of Corollary 1) holds for the composition (X, B) of each set of elements $\{(X_i, \mathsf{B}_i)\}_{i=1}^N$ where (X_i, B_i) is such that the condition (i) of Corollary 1 holds with respect to $\mathcal{C}_i = (\mathscr{A}_i, \mathsf{G}_i, X_i)$.

Example 2a. Consider that a composition structure $\mathfrak{D}_a$ of a two level contract hierarchy $(\mathcal{C}'_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}'_{bat}, \mathcal{C}_{lMeter}\})$ is the resulting composition structure from making the following modifications to the composition structure and two-level contract hierarchy shown in Figure 8: in the contract $\mathcal{C}'_{bat} = (\{\}, \mathsf{G}'_{bat}, X_{bat})$, it holds that $\mathsf{G}'_{bat} = \Omega$, instead of being specified by the equation $v_{ref} - v_{gnd} = 5V$; A'_{Esys} in the contract $\mathcal{C}'_{Esys} = (\{\mathsf{A}'_{Esys}\}, \mathsf{G}_{Esys}, X_{Esys})$ is specified by the equations $f = h$ and $v_{ref} - v_{gnd} = 5V$, instead of only $f = h$; and that an outgoing arc has been added from A'_{Esys} to A_{pot1}. In accordance with the relation (18), the intent is that $\mathsf{A}'_{Esys} \cap \Omega \subseteq \mathsf{A}_{pot1}$, which is equal to $\emptyset \neq \mathsf{A}'_{Esys} \subseteq \mathsf{A}_{pot1}$. In accordance with Definition 5, due to the fact that A_{pot1} restricts $\{v_{ref}, v_{gnd}\}$, A_{Esys} also

needs to restrict $\{v_{ref}, v_{gnd}\}$ in order for the relation (18) to hold. Note that neither v_{ref} nor v_{gnd} are in X_{Esys}.

Now consider the set of elements $\{\mathbb{E}'_{bat}, \mathbb{E}_{pot}, \mathbb{E}_{lMeter}\}$ where $\mathbb{E}_{pot}$ and $\mathbb{E}_{lMeter}$ are shown in Figure 4 and where $\mathbb{E}'_{bat}$ is the modification of $\mathbb{E}_{bat}$ such that the behavior of $\mathbb{E}'_{bat}$ is equal to Ω, instead of being specified by the equation $v_{ref} - v_{gnd} = 5V$. It trivially holds that the elements $\mathbb{E}'_{bat}$, $\mathbb{E}_{pot}$, $\mathbb{E}_{lMeter}$ are such that the condition (i) of Corollary 1 holds with respect to $\mathcal{C}'_{bat}$, $\mathcal{C}_{pot}$, and $\mathcal{C}_{lMeter}$. Furthermore, it can easily be realized that the relations (17) and (18) hold for $\mathfrak{D}_a$. However, since the relation $v_{ref} - v_{gnd} = 5V$ is ensured by A'_{Esys}, rather than by the behavior of $\mathbb{E}'_{bat}$, the behavior of the composition $\mathbb{E}'_{sys}$ of $\{\mathbb{E}'_{bat}, \mathbb{E}_{pot}, \mathbb{E}_{lMeter}\}$ onto X_{Esys}, is Ω, rather than being specified by $l = h$. This means that $\mathbb{E}'_{sys}$ is not such that the relation (7) of the condition (i) of Corollary 1 holds with respect to $\mathcal{C}'_{Esys}$, i.e. it does not hold that $\mathsf{A}'_{Esys} \cap \mathsf{B}'_{Esys} \subseteq \mathsf{G}'_{Esys}$. Thus, the composition $\mathbb{E}'_{Esys}$ of $\{\mathbb{E}'_{bat}, \mathbb{E}_{pot}, \mathbb{E}_{lMeter}\}$ onto X_{Esys} is not such that the condition (i) of Corollary 1 holds with respect to $\mathcal{C}'_{Esys}$. □

As shown in Example 2a, the fact that it is necessary for A'_{Esys} to restrict port variables in $X_{bat} \cup X_{pot} \cup X_{lMeter} \setminus X_{Esys}$, in order for the relation (18) to hold, means that even if the relations (17) and (18) do hold, $(\mathcal{C}'_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}'_{bat}, \mathcal{C}_{lMeter}\})$ is not proper. Notably, regardless if is necessary or not for A'_{Esys} to restrict port variables in $X_{bat} \cup X_{pot} \cup X_{lMeter} \setminus X_{Esys}$, if A'_{Esys} does restrict such variables, then in accordance with Definition 5, there does not exists an element that non-trivially fulfills A'_{Esys}. That is, the fact that A'_{Esys} restricts variables in $X_{bat} \cup X_{pot} \cup X_{lMeter} \setminus X_{Esys}$ is undesirable all together.

Considering the general case with a composition structure $\mathfrak{D}$, a sufficient condition for ensuring that $\mathsf{A}_{\mathscr{A}}$ does not restrict port variables in $\bigcup_{i=1}^{N} X_i \setminus X$, is to ensure that $\mathsf{A}_{\mathscr{A}}$ does not constrain any port variable in $\bigcup_{i=1}^{N} X_i \setminus X$, i.e. that it holds that

$$(X_{\mathsf{A}_{\mathscr{A}}} \setminus X) \cap \bigcup_{i=1}^{N} X_i = \emptyset \ . \tag{19}$$

Notably, while ensuring that $\mathsf{A}_{\mathscr{A}}$ does not restrict port variables in $\bigcup_{i=1}^{N} X_i \setminus X$, in accordance with Proposition 4, it is also the case that the relation (19) can be enforced *without loosing expressiveness*. However, the fact that the relation 19 holds in combination with the relations (17)-(18), as will be shown by the following illustrative Examples 2b and 2c, this fact is not

sufficient to ensure that $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^{N})$ is proper; thus, additional conditions are presented in the following.

Example 2b. Suppose that a composition structure $\mathfrak{D}_b$ of a two level contract hierarchy $(\mathcal{C}''_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}''_{lMeter}\})$ is the resulting composition structure from making the following modifications to the composition structure and two-level contract hierarchy shown in Figure 8: both the guarantees G''_{lMeter} and G''_{Esys} in $\mathcal{C}''_{Esys} = (\{\mathsf{A}_{Esys}\}, \mathsf{G}''_{Esys}, X_{Esys})$ and $\mathcal{C}''_{lMeter} = (\{\mathsf{A}_{lMeter}\}, \mathsf{G}''_{lMeter}, X_{lMeter})$ are specified by the equations $\frac{v_{branch} - v_{gnd}}{5} = f$ and $f = l$, instead of $f = l$. Considering the composition of the set of elements $\{\mathbb{E}_{bat}, \mathbb{E}_{pot}, \mathbb{E}_{lMeter}\}$ onto X_{Esys}, i.e. the element $\mathbb{E}_{Esys}$ shown in Figure 4, due to the fact that $\mathsf{A}_{Esys} \cap \mathsf{B}_{Esys}$ is specified by the equations $f = h$ and $l = h$, it holds that $\mathsf{A}_{Esys} \cap \mathsf{B}_{Esys}$ is non-empty and constrains exactly $\{f, l, h\}$. In accordance with Definition 5, from the fact that G''_{Esys} does indeed restrict both v_{branch} and v_{gnd}, where neither of these are in $\{f, l, h\}$, it follows that the relation (7) of the condition (i) of Corollary 1 does not hold, i.e. that $\mathsf{A}_{Esys} \cap \mathsf{B}_{Esys} \nsubseteq \mathsf{G}''_{Esys}$. □

Example 2c. Consider that a composition structure $\mathfrak{D}_b$ of a two level contract hierarchy $(\mathcal{C}_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}'''_{lMeter}\})$ is the resulting composition structure from making the following modifications to the composition structure and two-level contract hierarchy shown in Figure 8: the guarantee G'''_{Esys} in the contract $\mathcal{C}'''_{lMeter} = (\{\mathsf{A}_{lMeter}\}, \mathsf{G}'''_{lMeter}, X_{lMeter})$ is specified by the relation $l - 0.1 \leq f \leq l + 0.1$, instead of the equation $f = l$. Considering the element $\mathbb{E}_{Esys}$, since B_{Esys} is specified by the equation $l = f$ and $\mathsf{A}_{Esys} \cap \mathsf{G}'''_{Esys}$ by the relation $l - 0.1 \leq f \leq l + 0.1$ and the equation $h = f$, the relation (8) of the condition (i) of Corollary 1 does not hold. That is, it does not hold that $\mathsf{A}_{Esys} \cap \mathsf{G}'''_{Esys} \subseteq \mathsf{B}_{Esys}$. □

In both Examples 2b and 2c, it trivially holds that the elements $\mathbb{E}_{bat}$, $\mathbb{E}_{pot}$, and $\mathbb{E}_{lMeter}$ are such that the condition (i) of Corollary 1 hold with respect to the contracts containing their sets of port variables. Furthermore, it can easily be realized that the relations (17) and (18) hold for both $\mathfrak{D}_b$ and $\mathfrak{D}_c$. However, since neither one of the relations $\mathsf{A}_{Esys} \cap \mathsf{B}_{Esys} \subseteq \mathsf{G}''_{Esys}$ and $\mathsf{A}_{Esys} \cap \mathsf{G}'''_{Esys} \subseteq \mathsf{B}_{Esys}$ hold, it does not follow that the composition $\mathbb{E}_{Esys}$ of $\{\mathbb{E}_{bat}, \mathbb{E}_{pot}, \mathbb{E}_{lMeter}\}$ onto X_{Esys} is such that the condition (i) of Corollary 1 holds with respect to any of $\mathcal{C}''_{Esys}$ or $\mathcal{C}'''_{Esys}$. Hence, despite the fact that the relations (17) and (18) hold for $\mathfrak{D}_b$ and $\mathfrak{D}_c$, neither $(\mathcal{C}''_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}''_{lMeter}\})$ nor $(\mathcal{C}_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}'''_{lMeter}\})$ are proper.

For a two level contract hierarchy $((\mathscr{A}, \mathsf{G}, X), \{(\{\mathsf{A}_{ij}\}_{j=1}^{M_i}, \mathsf{G}_i, X_i)\}_{i=1}^{N})$, similar to Example 2a, Example 2b shows that if G restricts a variable in $\bigcup_{i=1}^{N} X_i \setminus (X \cup X_{\mathsf{A}_{\mathscr{A}}})$, then $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^{N})$ is not proper. In accordance with Proposition 4, a sufficient condition to avoid this case, but without loosing expressiveness, is $(X_{\mathsf{G}} \setminus X_{\mathsf{A}_{\mathscr{A}}}) \cap \bigcup_{i=1}^{N} X_i = \emptyset$. This and the relation (19) imply that

$$(X_{\mathsf{G}} \setminus X) \cap \bigcup_{i=1}^{N} X_i = \emptyset \ . \tag{20}$$

Example 2c, on the other hand, shows that it is necessary for G to be a subset of the extended projection of the guarantees G_i onto $\Xi \setminus (\bigcup_{i=1}^{N} X_i \setminus X)$. That is, it is necessary that $\mathsf{G} \subseteq \widehat{proj}_{\Xi \setminus (\bigcup_{i=1}^{N} X_i \setminus X)}(\bigcap_{i=1}^{N} \mathsf{G}_i)$. However, given that this holds, as well the relations (19) and (20), it follows that G must be equal to $\widehat{proj}_{\Xi \setminus (\bigcup_{i=1}^{N} X_i \setminus X)}(\bigcap_{i=1}^{N} \mathsf{G}_i)$, rather than a subset, i.e. that

$$\mathsf{G} = \widehat{proj}_{\Xi \setminus (\bigcup_{i=1}^{N} X_i \setminus X)}(\bigcap_{i=1}^{N} \mathsf{G}_i) \ . \tag{21}$$

This can be realized by considering that $\widehat{proj}_{\Xi \setminus (\bigcup_{i=1}^{N} X_i \setminus X)}(\bigcap_{i=1}^{N} \mathsf{G}_i) \subseteq \mathsf{G}$, which follows from the fact that the relations (20) and (19) respectively imply that $\mathsf{G} = \widehat{proj}_{\Xi \setminus (\bigcup_{i=1}^{N} X_i \setminus X)}(\mathsf{G})$ in accordance with Proposition 1 and $\bigcap_{i=1}^{N} \mathsf{G}_i \subseteq \bigcap_{\mathsf{W} \in dPred(\mathsf{G}) \cap \{\mathsf{G}_i\}_{i=1}^{N}} \mathsf{W} \subseteq \mathsf{G}$.

To characterize a composition structure where the relations (17)-(21) hold, the concept of a *proper composition structure* is introduced.

Definition 10 (Proper Composition Structure)**.** Given a composition structure $\mathfrak{D}$ of a two-level contract hierarchy

$$((\mathscr{A}, \mathsf{G}, X), \{(\{\mathsf{A}_{ij}\}_{j=1}^{M_i}, \mathsf{G}_i, X_i)\}_{i=1}^{N})$$

consisting of a contract $(\mathscr{A}, \mathsf{G}, X)$ and set of contracts $\{(\{\mathsf{A}_{ij}\}_{j=1}^{M_i}, \mathsf{G}_i, X_i)\}_{i=1}^{N}$ where $X \subseteq \bigcup_{i=1}^{N} X_i$, *the composition structure* $\mathfrak{D}$ *is proper*, if it holds that:

i) $\bigcap_{\mathsf{W} \in dPred(\mathsf{G}) \cap \{\mathsf{G}_i\}_{i=1}^{N}} \mathsf{W} \subseteq \mathsf{G}$;

ii) $\bigcap_{\mathsf{W} \in dPred(\mathsf{A}_{ij})} \mathsf{W} \subseteq A_{ij}$ for each i, j;

iii) $((X_{\mathsf{G}} \cup X_{\mathsf{A}_{\mathscr{A}}}) \setminus X) \cap \bigcup_{i=1}^{N} X_i = \emptyset$; and

iv) $\mathsf{G} = \widehat{proj}_{\Xi\setminus(\bigcup_{i=1}^{N} X_i\setminus X)}(\bigcap_{i=1}^{N} \mathsf{G}_i)$. □

The condition (i) and (ii) of Definition 10 are the relations (17) and (18), respectively. The condition (iii) of Definition 10 combines the relations (19) and (20) into a single condition. Considering Example 2a, due to the fact that

$$((X_{\mathsf{G}_{Esys}} \cup X_{\mathsf{A}'_{Esys}}) \setminus X_{Esys}) \cap (X_{pot} \cup X_{bat} \cup X_{lMeter}) = \{v_{ref}, v_{gnd}\} \neq \emptyset,$$

the condition (iii) of Definition 10 ensures that the case highlighted in Example 2a is avoided. Furthermore, the condition (iii) also ensures that the case highlighted in Example 2b is avoided considering that

$$((X_{\mathsf{G}''_{Esys}} \cup X_{\mathsf{A}_{Esys}}) \setminus X_{Esys}) \cap (X_{pot} \cup X_{bat} \cup X_{lMeter}) = \{v_{branch}, v_{gnd}\} \neq \emptyset\ .$$

The condition (iv) of Definition 10 is the relation (21). Considering Example 2c, it holds that $\Xi \setminus \{v_{ref}, v_{gnd}, v_{branch}\} = \Xi \setminus ((X_{pot} \cup X_{bat} \cup X_{lMeter}) \setminus X_{Esys})$, which means that

$$\widehat{proj}_{\Xi\setminus((X_{pot}\cup X_{bat}\cup X_{lMeter})\setminus X_{Esys})}(\mathsf{G}'''_{lMeter} \cap \mathsf{G}_{bat} \cap \mathsf{G}_{pot})$$

and G_{Esys} are specified by the equation $f = l$ and the relation $l - 0.1 \leq f \leq l + 0.1$, respectively. It follows that

$$\mathsf{G}_{Esys} \not\subseteq \widehat{proj}_{\Xi\setminus((X_{pot}\cup X_{bat}\cup X_{lMeter})\setminus X_{Esys})}(\mathsf{G}'''_{lMeter} \cap \mathsf{G}_{bat} \cap \mathsf{G}_{pot})\ ,$$

which means that the condition (iv) of Definition 10 ensures that the case highlighted in Example 2c is avoided.

A theorem that presents sufficient conditions of a two-level contract hierarchy being proper now follows.

Theorem 5. *Given a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and a set of contracts $\{\mathcal{C}_i\}_{i=1}^{N}$ where $\mathcal{C}_i = (\mathscr{A}_i, \mathsf{G}_i, X_i)$ and $X \subseteq \bigcup_{i=1}^{N} X_i$, the two-level contract hierarchy $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^{N})$ is proper if there exists a proper composition structure of $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^{N})$.*

The proof of Theorem 5 is found in Appendix I.

To illustrate the use of Theorem 5, consider the scenario with the phases (I'-III') and, more specifically, that the two-level contract hierarchy established in the phase (I') is $(\mathcal{C}_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\})$. As previously

mentioned, in order to complete the phase (III'), this two-level contract hierarchy needs to be proper. As expressed in Theorem 5, a sufficient condition for this, is that there exists a proper composition structure of $(\mathcal{C}_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\})$. Hence, in accordance with Theorem 5, if e.g. the composition structure of $(\mathcal{C}_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\})$ in Figure 8 is proper, then $(\mathcal{C}_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\})$ must also be proper. The following will examine if the composition structure in Figure 8 is indeed proper, i.e. if it is in accordance with Definition 10.

As shown in the composition structure of $(\mathcal{C}_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\})$ in Figure 8, it trivially holds that the conditions (i) and (ii) of Definition 10 hold. Furthermore, considering that the union of the sets of variables $\{f, l\}$ and $\{f, h\}$, respectively constrained by G_{Esys} and A_{Esys}, and the set $(X_{pot} \cup X_{bat} \cup X_{lMeter}) \setminus X_{Esys} = \{v_{ref}, v_{gnd}, v_{branch}\}$ are disjoint, i.e. that $\{f, l, h\} \cap \{v_{ref}, v_{gnd}, v_{branch}\} = \emptyset$, the condition (iii) of Definition 10 is met. Finally, since the extended projection of $\mathsf{G}_{lMeter} \cap \mathsf{G}_{bat} \cap \mathsf{G}_{pot}$ onto $\Xi \setminus \{v_{ref}, v_{gnd}, v_{branch}\} = \Xi \setminus ((X_{pot} \cup X_{bat} \cup X_{lMeter}) \setminus X_{Esys})$ is equal to G_{Esys}, the condition (iv) of Definition 10 also holds. This means that the composition structure shown in Figure 8 is proper.

Due to the fact the composition structure in Figure 8 is proper, it follows that $(\mathcal{C}_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\})$ also is proper in accordance with Theorem 5. Thus, in accordance with Definition 8, for any set of elements with sets of port variables X_{pot}, X_{bat}, X_{lMeter} where these elements are developed by the suppliers in the phase (II') such that the condition (i) of Corollary 1 holds with respect to the contracts $\mathcal{C}_{pot}$, $\mathcal{C}_{bat}$, and $\mathcal{C}_{lMeter}$, it follows that the composition of such a set of elements is such that the condition (i) of Corollary 1 holds with respect to $\mathcal{C}_{Esys}$. For example, consider that the suppliers develop the elements $\mathbb{E}_{bat}$, $\mathbb{E}_{pot}$, and $\mathbb{E}_{lMeter}$ in the phase (II'). It trivially holds that each element $\mathbb{E}_i \in \{\mathbb{E}_{bat}, \mathbb{E}_{pot}, \mathbb{E}_{lMeter}\}$ is such that the condition (i) of Corollary 1 holds with respect to $\mathcal{C}_i \in \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\}$. Due to this and the fact that $(\mathcal{C}_{Esys}, \{\mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\})$ is proper, it automatically follows that $\mathbb{E}_{Esys}$ is such that the condition (i) of Corollary 1 holds with respect to $\mathcal{C}_{Esys}$, as desired in the phase (III').

7 Related work

In Section 1- 6, a vast number of general assume-guarantee theories [4, 31–57] have been referred to, without a proper introduction. Therefore,

the following two paragraphs of this section is dedicated to describing the contexts and applications of these general theories, as well as related concrete theories. The rest of this section compares the present paper with these theories and other related work, focusing on technical matters that have not been previously discussed in the present paper.

As mentioned in Section 3, the notion of contracts was first introduced in [1] to be used as formal specification in object-oriented programming. Since then, the use of contracts has been extended to component-based design [83] and a contract theory for analog systems have been proposed in [36, 84]. Contracts have also been introduced in the formalisms *Behavior Interaction Priority (BIP)* [85] and *refinement calculus* [86], in [37, 87] and [38], respectively. Furthermore, in the european research project SPEEDS [6], a contract theory [3–5] was introduced as a means to meet the challenges in the design of heterogeneous systems [7–9]. Similar work to [3–5] is presented in [44] and in [33] with tool support [88], and also in a more applied setting in [89, 90]. The use of the theory [3–5] has been advocated in [10–15] and the use of contracts in general has been proposed for analyzes integration [91] and as a means to achieve functional safety in [92, 93] and also in [94] with tool support [95]. Contract theory has also been extended both with *modalities* [96] in [12, 39] and to a stochastic setting in [35, 40]. Meta theories of contracts have been established in [31, 34], and in [42] with refinement [43]. The theory [42] also clarifies the distinction between contracts and *specification theories*, e.g. [54, 97, 98] that extend interface automata [99, 100] and where [54] is shown to also support assume-guarantee reasoning in [101].

More generally than contracts, assume-guarantee reasoning can be traced back to two independent theories [102, 103] and [48] concerning *compositional* [29, 30] proof methods for concurrent programs. However, the ideas in [48, 102, 103] can be traced even further back to proof methods [2, 104, 105] for sequential programs and non-compositional proof methods [106–108] for concurrent programs. Since the conception of [48, 102, 103], several theories that extend the ideas in [102, 103] and [48], have emerged, such as e.g. [32, 46] and [47, 109], respectively. Furthermore, assume-guarantee reasoning has also been used in formal verification, see e.g. [57, 110, 111] or [112] for an overview. Moreover, automatic techniques for assume-guarantee reasoning have been proposed, see e.g. [113] or [114] for a survey. Given that the two approaches [102, 103] and [48] are the same in principle, meta theories [45, 49–52], and [52] as an extension of [115], have been introduced to unify [102, 103] and [48]. General assume-guarantee theories are also

presented in [55, 56], and in [41] based on [116, 117]. Furthermore, with inspiration from [46, 118], how to perform compositional verification on architecture models is described in [119].

In accordance with [3–5] and also with [14, 62], contracts and behaviors of elements are in the present paper both defined by relying on the concept of assertions as a set of runs. The concept of runs is, in turn, largely inspired by the works in [34, 120, 121] that generalize the concept of *traces* [59–61] to behaviors that are independent of a particular model of computation. Notably, two assertions are equivalent if they have the same runs, i.e. they are *trace-equivalent* [122], which means that assertions are limited to a weaker form of equivalence than e.g. observation equivalence [123] or equivalence through alternating simulation [124], which can be verified on labeled and alternating transition systems, respectively. However, as shown in [122], for deterministic models, trace equivalence means observation equivalence, and vice versa. This means that

In Section 2.2, the concept of an element that essentially corresponds to a HRC [67, 68] as used in [3–5], was introduced. The main difference is that a HRC can have several *implementations*, i.e. behaviors, which means that an element corresponds to an implementation of a HRC, rather than to a HRC itself. An element in the present paper is in that sense more similar to a component as defined in [63] that is inspired by the *tagged signal model* [125] and *interface theory* [126].

Notably, other assume-guarantee theories [3–5, 31, 35] have established operators on contracts, namely *parallel composition* [3–5, 31, 35, 42, 54], *conjunction* [3–5, 31, 35, 54], and *quotient* [31, 54]. While both parallel composition and conjunction merge a set of contracts into a single contract, the latter is only applicable when the contracts contain the same set of port variables and concerns the case when the contracts are specified for different *viewpoints* [127, 128]. Quotient computes a missing contract in a contract hierarchy to achieve compositionality. In accordance with [42] and the meta theory in [31], and as will be shown in Remark 2 in the following, these operators can be derived by introducing a *refinement preorder on contracts*; in the context of the present paper, this would mean that a contract $(\mathscr{A}', \mathsf{G}', X)$ *refines* another contract $(\mathscr{A}, \mathsf{G}, X)$ if each element (X, B) is such that: the condition (i) of Corollary 1 holds with respect to $(\mathscr{A}, \mathsf{G}, X)$ if the condition (i) of Corollary 1 holds with respect to $(\mathscr{A}', \mathsf{G}', X)$.

Remark 2 (Deriving Parallel Composition, Conjunction, and Qoutient). A

contract is a *parallel composition of a set of contracts* $\{(\mathscr{A}_i, \mathsf{G}_i, X_i)\}_{i=1}^N$ *onto a set of variables* $X \subseteq \bigcup_{i=1}^N X_i$ if the contract is a most refined contract $(\mathscr{A}, \mathsf{G}, X)$ such that $((\mathscr{A}, \mathsf{G}, X), \{(\mathscr{A}_i, \mathsf{G}_i, X_i)\}_{i=1}^N)$ is proper. A contract is a *conjunction of a set of contracts* $\{(\mathscr{A}_j, \mathsf{G}_j, X)\}_{j=1}^M$ *onto* $X \subseteq \bigcup_{i=1}^N X_i$ if the contract is a most refined contract $(\mathscr{A}, \mathsf{G}, X)$ such that, for each element (X, B), it holds that: $\mathsf{A}_{\mathscr{A}_i} \cap \mathsf{B} \subseteq \mathsf{G}_i$ and $\mathsf{A}_{\mathscr{A}_i} \cap \mathsf{G}_i \subseteq \mathsf{B}$ for each i, if $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ and $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$. Given a contract $(\mathscr{A}, \mathsf{G}, X)$, a set of contracts $\{(\mathscr{A}_i, \mathsf{G}_i, X_i)\}_{i=1}^N$, and a set of variables X' such that $X \subseteq X' \cup \bigcup_{i=1}^N X_i$, a contract is *a quotient of* $((\mathscr{A}, \mathsf{G}, X), \{(\mathscr{A}_i, \mathsf{G}_i, X_i)\}_{i=1}^N)$ *with respect to* X' if the contract is a least refined contract $(\mathscr{A}, \mathsf{G}, X')$ such that $(\mathscr{A}, \mathsf{G}, X)$ is a parallel composition of $\{(\mathscr{A}, \mathsf{G}, X')\} \cup \{(\mathscr{A}_i, \mathsf{G}_i, X_i)\}_{i=1}^N$ onto X. □

8 Conclusion

This paper has presented a general compositional contract theory for modeling and specifying *heterogeneous systems*. As the main contribution, given a contract for an element representing any part of a heterogeneous system, e.g. SW, mechanical, or electrical part, Corollary 1 presented clearly separated *conditions* (i) and (ii) on the element and its environment where the conditions ensure that the guarantee is non-trivially fulfilled by the composition of the element and the environment.

In contrast to similar conditions of other general assume-guarantee theories [3–5, 31–57], while explicitly considering the set of port variables of an element, the conditions (i) and (ii) require neither that this set is partitioned into inputs and outputs nor that the assumptions and guarantee must be specified over this set of port variables. The former means that the causality of the port variables can remain unspecified, which is common and recommended practice when modeling physical parts. The latter allows assigning the *responsibility* of fulfilling a global property to the element, which is necessary in order to properly express safety specifications for the element, e.g. in accordance with ISO 26262.

The ability to assign the *responsibility* of fulfilling a global property to an element, increases the expressiveness with respect to how a contract can be specified. To facilitate the specification of contracts in practice, scoping conditions were introduced that limit the set of port variables over which the assumptions and the guarantee of a contract are specified. These scoping conditions ensure that certain necessary properties of conditions (i)

and (ii) are are not violated without limiting expressiveness. Notably, these conditions can be checked, not only for the cases where the assumptions and the guarantee are specified using formal notation, but also when they are specified using *semi-formal notation*, e.g. as free text with formal references to port variables of elements. Hence, considering a tool where these checks are automatically performed, feedback to a user specifying a contract can be given, both when semi-formal and formal notations are used.

In the context of a scenario where a contract is used to outsource the development of an element, necessary contract properties *consistency* and *compatibility* were presented. Complementary necessary and sufficient conditions of these properties were also introduced where these conditions are easier to enforce in practice (e.g. by a tool) than their corresponding definitions. Furthermore, as a basis for structuring a two-level contract hierarchy in parallel to a composition of a set of elements, a graph, called a *composition structure*, was introduced. Based on a composition structure, sufficient conditions to achieve *compositionality* was presented. Note that proving that the sufficient conditions hold, requires specifying the assumptions and the guarantees using formal notation. However, regardless if the assumptions and guarantees are specified using formal, semi-formal, or *informal notation*, e.g. as free text, the conditions for structuring the overall intended relations between the assumptions and guarantees as a composition structure, still apply. This means that, regardless of the level of formalization used in the specifications, support for structuring a two-level contract hierarachy can be given in the form of a tool that enforces these conditions.

Considering the concepts presented above, they are all general in both the senses that they are relevant to any developer of heterogeneous system parts, and that they rely on a general set-theoretic formalism. Due to the generality of the theory, it is essentially applicable in any context, and it can also be instantiated by more concrete theories whenever needed. Moreover, the theory is tightly coupled with practical application where introduced definitions have been both well motivated by industrial needs and/or scenarios, and complemented with necessary and sufficient conditions that can more easily be enforced in practice, e.g. by tools implementing the theory. As previously mentioned, the concrete support that can be given by such tools, is not limited to the cases where formal notations are used, but also when semi-formal and informal notations are used. Hence, not only does the presented theory constitute a general compositional contract theory for modeling and specifying heterogeneous systems, but the theory is indeed also

accommodated for providing concrete support for developing such systems in practice.

Acknowledgements

This work is a result from a collaborative effort with automotive manufacturer *Scania CV AB*, and has received funding under grant agreement no.2011 – 04446 from *Fordonsstrategisk Forskning och Innovation* (FFI) - a partnership between the Swedish government and automotive industry for joint funding of research, innovation and development concentrating on Climate & Environment and Safety

References

[1] Meyer, B.: Applying "Design by Contract". Computer **25**(10) (October 1992) 40–51

[2] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. Commun. ACM **12**(10) (October 1969) 576–580

[3] Benveniste, A. et al.: Multiple Viewpoint Contract-Based Specification and Design. In de Boer, F. et al., eds.: Formal Methods for Components and Objects. Volume 5382 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2008) 200–225

[4] Benveniste, A., B. Caillaud, and R. Passerone: Multi-Viewpoint State Machines for Rich Component Models. In Nicolescu, G., and P. Mosterman, eds.: Model-Based Design for Embedded Systems. Taylor & Francis (2009) 487–518

[5] Sangiovanni-Vincentelli, A. L., W. Damm, and R. Passerone: Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. European Journal of Control **18**(3) (2012) 217–238

[6] SPEEDS: SPEculative and Exploratory Design in Systems Engineering (2006-2009)

[7] Henzinger, T., and J. Sifakis: The Discipline of Embedded Systems Design. Computer **40**(10) (Oct 2007) 32–40

[8] Lee, E.: Cyber Physical Systems: Design Challenges. In: Object Oriented Real-Time Distributed Computing (ISORC), 11th IEEE Int. Symp. on. (2008) 363–369

[9] Rawat, D. B., J. J. Rodrigues, and I. Stojmenovic: Cyber-Physical Systems: From Theory to Practice. CRC Press (2015)

[10] Baumgart, A. et al.: A Model-Based Design Methodology with Contracts to Enhance the Development Process of Safety-Critical Systems. In: Software Technologies for Embedded and Ubiquitous Systems. Volume 6399 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 59–70

[11] Damm, W., B. Josko, and T. Peinkamp: Contract Based ISO CD 26262 Safety Analysis. In: Safety-Critical Systems, 2009. SAE (2009)

[12] Damm, W. et al.: Using contract-based component specifications for virtual integration testing and architecture design. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2011. (March 2011) 1–6

[13] Westman, J., and M. Nyberg: 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, France (September 2013) NA

[14] Westman, J., M. Nyberg, and M. Törngren: 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, New York, NY, USA, Springer-Verlag New York, Inc. (2013) 166–177

[15] Westman, J., and M. Nyberg: Extending Contract theory with Safety Integrity Levels. In: HASE, 2015 IEEE 16th Int. Symposium on. (Jan 2015) 85–92

[16] Fritzson, P.: Introduction to modeling and simulation of technical and physical systems with Modelica. John Wiley & Sons (2011)

[17] Fritzson, P., and V. Engelson: Modelica - A unified object-oriented language for system modeling and simulation. In Jul, E., ed.: ECOO'98 - Object-Oriented Programming. Volume 1445 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1998) 67–90

[18] Fritzson, P.: Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach. John Wiley & Sons (2014)

[19] Erden, M. et al.: A review of function modeling: Approaches and applications. Artificial Intelligence for Engineering Design, Analysis and Manufacturing **22** (3 2008) 147–169

[20] Chandrasekaran, B., and J. Josephson: Function in Device Representation. Engineering with Computers **16**(3-4) (2000) 162–177

[21] Umeda, Y. et al.: Function, behaviour, and structure. Applications of artificial intelligence in engineering V **1** (1990) 177–194

[22] Liang, F. et al.: Model-based Requirement Verification : A case study. In: Proc. of the 9th Int. Modelica Conf. (2012) 263–268

[23] Schamai, W. et al.: Towards unified system modeling and simulation with modelicaml: Modeling of executable behavior using graphical notations. In: 7th Modelica Conference 2009, University Electronic Press (2009)

[24] Boulanger, J.-L., and V. Q. Dao: Requirements engineering in a model-based methodology for embedded automotive software. In: Research, Innovation and Vision for the Future, 2008. RIVF 2008. IEEE Int. Conf. on. (July 2008) 263–268

[25] Friedenthal, S., A. Moore, and R. Steiner: A Practical Guide to SysML: Systems Modeling Language. Morgan Kaufmann Inc., San Francisco, CA, USA (2008)

[26] IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems (2010)

[27] ISO 26262: Road vehicles-Functional safety (2011)

[28] Izosimov, V., U. Ingelsson, and A. Wallin: Requirement decomposition and testability in development of safety-critical automotive components,. In Ortmeier, F., and P. Daniel, eds.: Computer Safety, Reliability, and Security. Volume 7612 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 74–86

[29] de Roever, W., H. Langmaack, and A. Pnueli: Compositionality: The Significant Difference. Springer (1998)

[30] Hooman, J., and W. P. de Roever: The Quest Goes on: A Survey of Proofsystems for Partial Correctness of CSP. In: Current Trends in Concurrency, Overviews and Tutorials. Springer Berlin Heidelberg (1986) 343–395

[31] Benveniste, A. et al.: Contracts for System Design. Rapport de recherche RR-8147, INRIA (November 2012)

[32] Abadi, M., and L. Lamport: Composing specifications. ACM Trans. Program. Lang. Syst. **15**(1) (January 1993) 73–132

[33] Cimatti, A., and S. Tonetta: Contracts-refinement proof system for component-based embedded systems. Science of Computer Programming **97, Part 3** (2015) 333 – 348 Object-Oriented Programming and Systems (OOPS 2010)Modeling and Analysis of Compositional Software (papers from {EUROMICRO} SEAA'12).

[34] Negulescu, R.: Process Spaces. In: Proceedings of the 11th Int. Conf. on Concurrency Theory. CONCUR '00, London, UK, UK, Springer-Verlag (2000) 199–213

[35] Delahaye, B., B. Caillaud, and A. Legay: Probabilistic contracts: A compositional reasoning methodology for the design of systems with stochastic and/or non-deterministic aspects. Form. Methods Syst. Des. **38**(1) (February 2011) 1–32

[36] Sun, X. et al.: Contract-based System-Level Composition of Analog Circuits. In: Design Automation Conf., 2009. DAC '09. 46th ACM/IEEE. (july 2009) 605 –610

[37] Quinton, S., and S. Graf: Contract-based verification of hierarchical systems of components. In: Software Engineering and Formal Methods, 2008. SEFM '08. Sixth IEEE International Conference on. (nov. 2008) 377 –381

[38] Back, R.-J., and J. von Wright: Contracts, games, and refinement. Inf. Comput. **156**(1-2) (January 2000) 25–45

[39] Goessler, G., and J.-B. Raclet: Modal contracts for component-based design. In: Proc. of the 2009 7th IEEE Int. Conf. on Software Eng. and Formal Methods. SEFM '09, Washington, DC, USA, IEEE Computer Society (2009) 295–303

[40] Goessler, G., D. Xu, and A. Girault: Probabilistic contracts for component-based design. Formal Methods in System Design **41**(2) (2012) 211–231

[41] Broy, M.: Towards a theory of architectural contracts: - schemes and patterns of assumption/promise based system specification. In Broy, M., C. Leuxner, and T. Hoare, eds.: Software and Systems Safety - Specification and Verification. Volume 30 of NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press (2011) 33–87

[42] Bauer, S. et al.: Moving from specifications to contracts in component-based design. In Lara, J., and A. Zisman, eds.: Fundamental Approaches to Software Eng. Volume 7212 of Lec. Notes in Computer Science. Springer Berlin Heidelberg (2012) 43–58

[43] Bauer, S. S., R. Hennicker, and A. Legay: A meta-theory for component interfaces with contracts on ports. Sci. Comput. Program. **91** (2014) 70–89

[44] Le, T. T. H. et al.: A tag contract framework for modeling heterogeneous systems. Science of Computer Programming (2015) –

[45] Maier, P.: A set-theoretic framework for assume-guarantee reasoning. In Orejas, F., P. Spirakis, and J. van Leeuwen, eds.: Automata, Languages and Programming. Volume 2076 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2001) 821–834

[46] Mcmillan, K. L.: 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-Verlag (1999) 342–345

[47] Abadi, M., and L. Lamport: Conjoining specifications. ACM Trans. Program. Lang. Syst. **17**(3) (May 1995) 507–535

[48] Misra, J., and K. Chandy: Proofs of Networks of Processes. Software Engineering, IEEE Transactions on **SE-7**(4) (1981) 417–426

[49] Cau, A., and P. Collette: Parallel composition of assumption-commitment specifications. Acta Informatica **33**(2) (1996) 153–176

[50] Xu, Q., A. Cau, and P. Collette: On unifying assumption-commitment style proof rules for concurrency. In Jonsson, B., and J. Parrow, eds.: CONCUR'94: Concurrency Theory. Volume 836 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1994) 267–282

[51] Viswanathan, M., and R. Viswanathan: Foundations for Circular Compositional Reasoning. In Orejas, F., P. Spirakis, and J. van Leeuwen, eds.: Automata, Languages and Programming. Volume 2076 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2001) 835–847

[52] Tsay, Y.-K.: Compositional Verification in Linear-Time Temporal Logic. In Tiuryn, J., ed.: Foundations of Software Science and Computation Structures. Volume 1784 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2000) 344–358

[53] Amla, N. et al.: Abstract patterns of compositional reasoning. In Amadio, R., and D. Lugiez, eds.: CONCUR 2003 - Concurrency Theory. Volume 2761 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2003) 431–445

[54] Chilton, C., B. Jonsson, and M. Kwiatkowska: An algebraic theory of interface automata. Theoretical Computer Science **549** (September 2014) 146–174

[55] Tripakis, S. et al.: A Theory of Synchronous Relational Interfaces. ACM Trans. Program. Lang. Syst. **33**(4) (July 2011) 14:1–14:41

[56] Alur, R., and T. Henzinger: Reactive Modules. Formal Methods in System Design **15**(1) (1999) 7–48

[57] Grumberg, O., and D. E. Long: Model Checking and Modular Verification. ACM Trans. Program. Lang. Syst. **16**(3) (May 1994) 843–871

[58] Davis, M.: Infinite Games with Perfect Information. University of California, Berkeley (1961)

[59] Dill, D. L.: Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. In: Proceedings of the fifth MIT conference on Advanced research in VLSI, Cambridge, MA, USA, MIT Press (1988) 51–65

[60] Wolf, E. S.: Hierarchical Models of Synchronous Circuits for Formal Verification and Substitution. PhD thesis, Stanford University, Stanford, CA, USA (1996) UMI Order No. GAX96-12052.

[61] Brookes, S. D., C. A. R. Hoare, and A. W. Roscoe: A Theory of Communicating Sequential Processes. J. ACM **31**(3) (June 1984) 560–599

[62] Westman, J., and M. Nyberg: Environment-Centric Contracts for Design of Cyber-Physical Systems. In Dingel, J. et al., eds.: Model-Driven Engineering Languages and Systems. Volume 8767 of Lecture Notes in Computer Science. Springer International Publishing (2014) 218–234

[63] Simko, G. et al.: Towards a theory for cyber-physical systems modeling. In: Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems. CyPhy '14, New York, NY, USA, ACM (2014) 56–61

[64] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. Commun. ACM **13**(6) (June 1970) 377–387

[65] Lamport, L.: A Simple Approach to Specifying Concurrent Systems. Commun. ACM **32**(1) (January 1989) 32–45

[66] Abadi, M., and L. Lamport: The Existence of Refinement Mappings. Theor. Comput. Sci. **82**(2) (May 1991) 253–284

[67] Josko, B., Q. Ma, and A. Metzner: Designing Embedded Systems using Heterogeneous Rich Components. In: Proceedings of the INCOSE International Symposium. (2008)

[68] Damm, W.: Controlling Speculative Design Processes Using Rich Component Models. In: Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on. (june 2005) 118 – 119

[69] Westman, J., and M. Nyberg: Formal Architecture Modeling of Sequential C-Programs. In: Formal Aspects of Component Software. Lecture Notes in Computer Science. Springer International Publishing (2015)

[70] Gacek, A. et al.: Towards Realizability Checking of Contracts Using Theories. In Havelund, K., G. Holzmann, and R. Joshi, eds.: NASA Formal Methods. Volume 9058 of Lecture Notes in Computer Science. Springer International Publishing (2015) 173–187

[71] Le, T. T. H., and R. Passerone: Refinement-based synthesis of correct contract model decompositions. In: Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on. (Oct 2014) 134–143

[72] de Roever, W.-P.: The need for compositional proof systems: A survey. In de Roever, W.-P., H. Langmaack, and A. Pnueli, eds.: Compositionality: The Significant Difference. Volume 1536 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1998) 1–22

[73] Yu, E.: Towards modelling and reasoning support for early-phase requirements engineering. In: Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on. (Jan 1997) 226–235

[74] van Lamsweerde, A., and E. Letier: From object orientation to goal orientation: A paradigm shift for requirements engineering. In Wirsing, M., A. Knapp, and S. Balsamo, eds.: Radical Innovations of Software

and Systems Engineering in the Future. Volume 2941 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2004) 325–340

[75] Lapouchnian, A.: Goal-oriented requirements engineering: An overview of the current research. University of Toronto (2005)

[76] Jackson, M.: The world and the machine. In: Proceedings of the 17th International Conference on Software Engineering. ICSE '95, New York, NY, USA, ACM (1995) 283–292

[77] Jackson, M.: Software Requirements &Amp; Specifications: A Lexicon of Practice, Principles and Prejudices. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1995)

[78] Parnas, D. L.: Functional documents for computer systems. Science of Computer Programming **25** (1995) 41–61

[79] Nyberg, M.: Failure propagation modeling for safety analysis using causal bayesian networks. In: Control and Fault-Tolerant Systems (SysTol), 2013 Conference on. (Oct 2013) 91–97

[80] Nyberg, M., and J. Westman: Failure Propagation Modeling Based on Contracts Theory. In: Dependable Computing Conference (EDCC), 2015 Eleventh European. (Sept 2015) 108–119

[81] Namjoshi, K. S., and R. J. Trefler: On the completeness of compositional reasoning methods. ACM Trans. Comput. Logic **11**(3) (May 2010) 16:1–16:22

[82] Maier, P.: Compositional circular assume-guarantee rules cannot be sound and complete. In Gordon, A., ed.: Foundations of Software Science and Computation Structures. Volume 2620 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2003) 343–357

[83] Giese, H.: Contract-based Component System Design. In: Thirty-Third Annual Hawaii Int. Conf. on System Sciences (HICSS-33), Maui, IEEE Press (2000)

[84] Sun, X.: Compositional Design of Analog Systems Using Contracts. PhD thesis, EECS Department, University of California, Berkeley (May 2011)

[85] Bliudze, S., and J. Sifakis: The algebra of connectors: Structuring interaction in bip. In: Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software. EMSOFT '07, New York, NY, USA, ACM (2007) 11–20

[86] Back, R.-J. J., A. Akademi, and J. V. Wright: Refinement Calculus: A Systematic Introduction. 1st edn. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1998)

[87] Graf, S., and S. Quinton: Contracts for bip: Hierarchical interaction models for compositional verification. In: Proceedings of the 27th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems. FORTE '07, Berlin, Heidelberg, Springer-Verlag (2007) 1–18

[88] Cimatti, A., M. Dorigatti, and S. Tonetta: Ocra: A tool for checking the refinement of temporal contracts. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. (Nov 2013) 702–705

[89] Derler, P. et al.: Cyber-physical system design contracts. In: ICCPS '13: ACM/IEEE 4th International Conference on Cyber-Physical Systems. (2013)

[90] Törngren, M. et al.: Design Contracts for Cyber-Physical Systems: Making Timing Assumptions Explicit. Technical Report UCB/EECS-2012-191, EECS Department, University of California, Berkeley (Aug 2012)

[91] Ruchkin, I. et al.: Contract-based integration of cyber-physical analyses. In: Proceedings of the 14th International Conference on Embedded Software. EMSOFT '14, New York, NY, USA, ACM (2014) 23:1–23:10

[92] Bate, I., R. Hawkins, and J. McDermid: A contract-based approach to designing safe systems. In: Proc. of the 8th Australian Workshop on Safety Critical Sys. and SW - Volume 33. SCS '03, Australian Computer Society, Inc. (2003) 25–36

[93] Arts, T., M. Dorigatti, and S. Tonetta: Making implicit safety requirements explicit. In Bondavalli, A., and F. Di Giandomenico, eds.:

Computer Safety, Reliability, and Security. Volume 8666 of Lecture Notes in Computer Science. Springer International Publishing (2014) 81–92

[94] Soderberg, A., and R. Johansson: Safety contract based design of software components. In: Software Reliability Engineering Workshops (ISSREW), 2013 IEEE Int. Symposium on. (Nov 2013) 365–370

[95] Soderberg, A., and B. Vedder: Composable safety-critical systems based on pre-certified software components. In: Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on. (Nov 2012) 343–348

[96] Larsen, K.: Modal specifications. In Sifakis, J., ed.: Automatic Verification Methods for Finite State Systems. Volume 407 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1990) 232–246

[97] David, A. et al.: Timed i/o automata: A complete specification theory for real-time systems. In: Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control. HSCC '10, New York, NY, USA, ACM (2010) 91–100

[98] Raclet, J.-B. et al.: A modal interface theory for component-based design. Fundam. Inf. **108**(1-2) (January 2011) 119–149

[99] de Alfaro, L., and T. A. Henzinger: Interface automata. SIGSOFT Softw. Eng. Notes **26**(5) (September 2001) 109–120

[100] Lynch, N. A., and M. R. Tuttle: An introduction to input/output automata. CWI Quarterly **2** (1989) 219–246

[101] Chilton, C., B. Jonsson, and M. Kwiatkowska: Compositional assume-guarantee reasoning for input/output component theories. Science of Computer Programming **91, Part A** (2014) 115 – 137 Special Issue on Formal Aspects of Component Software (Selected Papers from FACS'12).

[102] Jones, C. B.: Specification and Design of (Parallel) Programs. In Mason, R. E. A., ed.: Information Processing 83. Volume 9 of IFIP Congress Series., Paris, France, IFIP, North-Holland (September 1983) 321–332

[103] Jones, C. B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4) (October 1983) 596–619

[104] Floyd, R. W.: Assigning meanings to programs. In Schwartz, J. T., ed.: Mathematical Aspects of Computer Science. Volume 19 of Proceedings of Symposia in Applied Mathematics., Providence, Rhode Island, American Mathematical Society (1967) 19–32

[105] Dijkstra, E. W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Commun. ACM **18**(8) (August 1975) 453–457

[106] Owicki, S., and D. Gries: An axiomatic proof technique for parallel programs i. Acta Informatica **6**(4) (1976) 319–340

[107] Ashcroft, E. A.: Proving assertions about parallel programs. J. Comput. Syst. Sci. **10**(1) (February 1975) 110–135

[108] Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **3**(2) (March 1977) 125–143

[109] Pnueli, A.: In transition from global to modular temporal reasoning about programs. In Apt, K., ed.: Logics and Models of Concurrent Systems. Volume 13 of NATO ASI Series. Springer Berlin Heidelberg (1985) 123–144

[110] Alur, R. et al.: Mocha: Modularity in model checking. In Hu, A., and M. Vardi, eds.: Computer Aided Verification. Volume 1427 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1998) 521–525

[111] Gurov, D., M. Huisman, and C. Sprenger: Compositional Verification of Sequential Programs with Procedures. Inf. Comput. **206**(7) (July 2008) 840–868

[112] Kupferman, O., and M. Y. Vardi: An automata-theoretic approach to modular model checking. ACM Trans. Program. Lang. Syst. **22**(1) (January 2000) 87–128

[113] Păsăreanu, C. S. et al.: Learning to divide and conquer: Applying the l* algorithm to automate assume-guarantee reasoning. Form. Methods Syst. Des. **32**(3) (June 2008) 175–205

[114] Cobleigh, J. M., G. S. Avrunin, and L. A. Clarke: Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. ACM Trans. Softw. Eng. Methodol. **17**(2) (May 2008) 7:1–7:52

[115] Jonsson, B., and T. Yih-Kuen: Assumptionguarantee specifications in linear-time temporal logic. Theoretical Computer Science **167**(1) (1996) 47 – 72

[116] Broy, M.: Compositional refinement of interactive systems. J. ACM **44**(6) (November 1997) 850–891

[117] Broy, M., and K. Stølen: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2001)

[118] Hammond, J., R. Rawlings, and A. Hall: Will it work? [requirements engineering]. In: Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on. (2001) 102–109

[119] Cofer, D. et al.: Compositional verification of architectural models. In: Proceedings of the 4th International Conference on NASA Formal Methods. NFM'12, Berlin, Heidelberg, Springer-Verlag (2012) 126–140

[120] Passerone, R.: Semantic Foundations for Heterogeneous Systems. PhD thesis, University of California, Berkeley (2004) AAI3146975.

[121] Burch, J., R. Passerone, and A. Sangiovanni-Vincentelli: Overcoming heterophobia: modeling concurrency in heterogeneous systems. In: Application of Concurrency to System Design, 2001. Proceedings. 2001 International Conference on. (2001) 13–32

[122] Engelfriet, J.: Determinancy (observation equivalence = trace equivalence). Theoretical Computer Science **36** (1985) 21 – 25

[123] Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1982)

[124] Alur, R. et al.: Alternating refinement relations. In: In Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR'98), volume 1466 of LNCS, Springer-Verlag (1998) 163–178

[125] Lee, E., and A. Sangiovanni-Vincentelli: A framework for comparing models of computation. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **17**(12) (dec 1998) 1217 –1229

[126] de Alfaro, L., and T. A. Henzinger: Interface theories for component-based design. In Henzinger, T. A., and C. M. Kirsch, eds.: Embedded Software. Volume 2211 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2001) 148–165

[127] Persson, M. et al.: A characterization of integrated multi-view modeling in the context of embedded and cyber-physical systems. In: Proceedings of the Eleventh ACM International Conference on Embedded Software. EMSOFT '13, Piscataway, NJ, USA, IEEE Press (2013) 10:1–10:10

[128] Törngren, M. et al.: Integrating viewpoints in the development of mechatronic products. Mechatronics **24**(7) (2014) 745 – 762 1. Model-Based Mechatronic System Design 2. Model Based Engineering.

Appendix I Proofs

This section presents proofs of Propositions 1 and 5, Lemmas 1 and 2, and Theorems 2 and 5.

Proposition 1. *Given an assertion* W*, it holds that a set of variables* X *is equal to* X_{W} *if and only if each variable in* X *is constrained by* W *and* $\widehat{proj}_X(\mathsf{W}) = \mathsf{W}$.

Proof. Consider an assertion W.

For the if part, assume that there exists a set of variables X constrained by W such that $\widehat{proj}_X(\mathsf{W}) = \mathsf{W}$. Assume $X \neq X_{\mathsf{W}}$, will be shown to lead to a contradiction. This means that it either holds that: i) $X = \emptyset$ and $X_{\mathsf{W}} \neq \emptyset$ or ii) there exists a variable $x \in X$ that is not also in X_{W}. From assuming that the case (i) holds, in accordance with the relation (2), it follows that $\mathsf{W} = \Omega$ since $\widehat{proj}_{X=\emptyset}(\mathsf{W}) = \Omega$ and $\widehat{proj}_X(\mathsf{W}) = \mathsf{W}$. This and the fact that $X_{\mathsf{W}} \neq \emptyset$ imply, in accordance with Definition 1, that there exists a variable $x \in X_{\mathsf{W}}$ such that $\widehat{proj}_{\Xi\setminus\{x\}}(\Omega) \neq \Omega$. This is a contradiction since, in accordance with Section 2.1 and the relation (2), Ω is indeed obtained if the set of all runs for $\Xi \setminus \{x\}$ is extended with all runs for x. Hence, the case (i) must be false, and it must either be that the case (ii) holds or that the assumption $X \neq X_{\mathsf{W}}$ is false. Assuming that the case (ii) holds directly yields a contradiction since this would mean that there exists a variable $x \notin X_{\mathsf{W}}$ such that $\widehat{proj}_{\Xi\setminus\{x\}}(\mathsf{W}) \neq \mathsf{W}$, which is not in accordance with Definition 1. This means that the assumption that $X \neq X_{\mathsf{W}}$ must be false and thus, it follows that $X = X_{\mathsf{W}}$.

For the if-only part, it suffices to show that $\widehat{proj}_{X_{\mathsf{W}}}(\mathsf{W}) = \mathsf{W}$, since W constrains each variable in X_{W} in accordance with Definition 1. Assume that $\widehat{proj}_{X_{\mathsf{W}}}(\mathsf{W}) \neq \mathsf{W}$, which will be shown to lead to a contradiction. This implies that that there exists a run $\omega_{\Xi,T} \in \mathsf{W}$ and a pair $(x \notin X_{\mathsf{W}}, \xi')$ such that a run $\omega'_{\Xi,T}$ is not in W if $\omega'_{\Xi,T}$ is obtained by replacing the pair $(x, \xi) \in \omega_{\Xi,T}$ with (x, ξ'). In accordance with the relation (2), this means that $\widehat{proj}_{\Xi\setminus\{x\}}(\mathsf{W}) \neq \mathsf{W}$ and, thus, W constrains x in accordance with Definition 1. However, this contradicts the fact that $x \notin X_{\mathsf{W}}$, which means that the assumption that $\widehat{proj}_{X_{\mathsf{W}}}(\mathsf{W}) \neq \mathsf{W}$ must be false, and it rather must hold that $\widehat{proj}_{X_{\mathsf{W}}}(\mathsf{W}) = \mathsf{W}$, which completes the proof. □

Lemma 1. *Given two assertions* W *and* W' *where* $\mathsf{W} \cup \mathsf{W}' \neq \emptyset$*, it holds that* $X_{\mathsf{W}\cup\mathsf{W}'} \subseteq X_{\mathsf{W}} \cup X_{\mathsf{W}'}$.

Proof. In accordance with Proposition 1, it holds that

$$\mathsf{W} \cup \mathsf{W}' = \widehat{proj}_{X_\mathsf{W}}(\mathsf{W}) \cup \widehat{proj}_{X_{\mathsf{W}'}}(\mathsf{W}') \ .$$

Furthermore, in accordance with the relation (2), it holds that $\widehat{proj}_{X_\mathsf{W}}(\mathsf{W})$ and $\widehat{proj}_{X_{\mathsf{W}'}}(\mathsf{W}')$ are obtained by extending $proj_{X_\mathsf{W}}(\mathsf{W})$ and $proj_{X_{\mathsf{W}'}}(\mathsf{W}')$ with all possible runs for $\Xi \setminus X_\mathsf{W}$ and $\Xi \setminus X_{\mathsf{W}'}$, respectively. Notably, both these set of runs are extended with all possible runs for $\Xi \setminus (X_\mathsf{W} \cup X_{\mathsf{W}'})$. This means, since $\mathsf{W} \cup \mathsf{W}' = \widehat{proj}_{X_\mathsf{W}}(\mathsf{W}) \cup \widehat{proj}_{X_{\mathsf{W}'}}(\mathsf{W}')$, that it follows that $\mathsf{W} \cup \mathsf{W}'$ is obtained by extending $proj_{X_\mathsf{W} \cup X_{\mathsf{W}'}}(\mathsf{W} \cup \mathsf{W}')$ with all possible runs for $\Xi \setminus (X_\mathsf{W} \cup X_{\mathsf{W}'})$. In accordance with Definition 1, this means that $X_{\mathsf{W} \cup \mathsf{W}'} \subseteq X_\mathsf{W} \cup X_{\mathsf{W}'}$. □

Proposition 5. *Given two assertions* W *and* W' *where* $\mathsf{W} \cap \mathsf{W}' \neq \emptyset$*, it holds that* $X_{\mathsf{W} \cap \mathsf{W}'} \subseteq X_\mathsf{W} \cup X_{\mathsf{W}'}$.

Proof. Given two assertions W and W' where $\mathsf{W} \cap \mathsf{W}' \neq \emptyset$, in accordance with the relation (1), for each run $\omega \in \mathsf{W} \cap \mathsf{W}'$, it holds that $proj_{X_\mathsf{W} \cup X_{\mathsf{W}'}}(\{\omega\})$ is in both $proj_{X_\mathsf{W} \cup X_{\mathsf{W}'}}(\mathsf{W})$ and $proj_{X_\mathsf{W} \cup X_{\mathsf{W}'}}(\mathsf{W}')$. This and since, in accordance with Proposition 1 and Section 2.1.2, it holds that $\widehat{proj}_{X_\mathsf{W} \cup X_{\mathsf{W}'}}(\mathsf{W}) = \mathsf{W}$ and $\widehat{proj}_{X_\mathsf{W} \cup X_{\mathsf{W}'}}(\mathsf{W}) = \mathsf{W}'$, imply that the run in $\widehat{proj}_{X_\mathsf{W} \cup X_{\mathsf{W}'}}(\{\omega\})$ must also be in both W and W'. Overall, this means that each run $\omega_{X_\mathsf{W} \cup X_{\mathsf{W}'}, T}$ in $proj_{X_\mathsf{W} \cup X_{\mathsf{W}'}}(\mathsf{W} \cap \mathsf{W}')$ can be extended with any run for $\Xi \setminus (X_\mathsf{W} \cup X_{\mathsf{W}'})$ over T, and the obtained run will also be in $\mathsf{W} \cap \mathsf{W}'$. In accordance with the relation (2), this means that $\widehat{proj}_{X_\mathsf{W} \cup X_{\mathsf{W}'}}(\mathsf{W} \cap \mathsf{W}') = \mathsf{W} \cap \mathsf{W}'$. In accordance with Proposition 1 and Section 2.1.2, it follows that $X_{\mathsf{W} \cap \mathsf{W}'} \subseteq X_\mathsf{W} \cup X_{\mathsf{W}'}$. □

Lemma 2. *Given two assertions* W *and* W'*, and a set of variables* X*, it holds that* $\widehat{proj}_X(\mathsf{W}) \subseteq \widehat{proj}_X(\mathsf{W}')$*, if* $\mathsf{W} \subseteq \mathsf{W}'$.

Proof. Given two assertions W and W' and a set of variables X, consider that $\mathsf{W} \subseteq \mathsf{W}'$. In accordance with the relation (1), the set of runs $proj_X(\mathsf{W})$ and $proj_X(\mathsf{W}')$ are obtained by removing each pair $(x \notin X, \xi)$ from each run in W and W', respectively. This and since $\mathsf{W} \subseteq \mathsf{W}'$, it follows that each pair that is removed from W' to obtain $proj_X(\mathsf{W}')$, is also removed from W to obtain $proj_X(\mathsf{W})$. Hence, it holds that $proj_X(\mathsf{W}) \subseteq proj_X(\mathsf{W}')$. In accordance with the relation (2), the assertions $\widehat{proj}_X(\mathsf{W})$ and $\widehat{proj}_X(\mathsf{W}')$ are obtained by extending each run $\omega_{\Xi,T}$ in $proj_X(\mathsf{W})$ and $proj_X(\mathsf{W}')$ with all possible runs for $\Xi \setminus X$ over T. This and since $proj_X(\mathsf{W}) \subseteq proj_X(\mathsf{W}')$,

it follows that each run $\omega_{\Xi,T}$ that is extended in $proj_X(\mathsf{W})$ with all possible runs for $\Xi \setminus X$ over T to obtain $\widehat{proj}_X(\mathsf{W})$, is also extended in $proj_X(\mathsf{W})$ with all possible runs for $\Xi \setminus X$ over T to obtain $\widehat{proj}_X(\mathsf{W}')$. Hence, it holds that $\widehat{proj}_X(\mathsf{W}) \subseteq \widehat{proj}_X(\mathsf{W}')$. □

Theorem 2. *Consider a contract* $(\mathscr{A}, \mathsf{G}, X)$ *and set of variables* $X_{Env_{\mathcal{E}}(\mathbb{E})}$. *It holds that there exists a contract* $(\mathscr{A}', \mathsf{G}', X)$ *where:*

a) $X_{\mathsf{A}_{\mathscr{A}'}} \subseteq X_{Env_{\mathcal{E}}(\mathbb{E})}$; *and*

b) $X_{\mathsf{G}'} \subseteq X_{Env_{\mathcal{E}}(\mathbb{E})} \cup X$

such that for each set of elements containing an element (X, B) *and where the pair* $(X_{Env_{\mathcal{E}}(\mathbb{E})}, \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})})$ *is the environment of* (X, B), *it holds that:*

i') $\mathsf{A}_{\mathscr{A}'} \cap \mathsf{B} \subseteq \mathsf{G}'$ *and* $\mathsf{A}_{\mathscr{A}'} \cap \mathsf{G}' \subseteq \mathsf{B}$, *and*

ii') $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}'}$ *and* $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G}' \neq \emptyset$,

if and only if

i) $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ *and* $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$, *and*

ii) $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ *and* $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$.

Proof. Consider a contract $(\mathscr{A}, \mathsf{G}, X)$ and a set of variables $X_{Env_{\mathcal{E}}(\mathbb{E})}$. First, let $(\mathscr{A}', \mathsf{G}', X)$ be a contract with a first property that $\mathsf{A}_{\mathscr{A}'}$ is the union of the behavior of each element $(X_{Env_{\mathcal{E}}(\mathbb{E})}, \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})})$ where $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}}$ and $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset$. This means, in accordance with Lemma 1 and Definition 2, it holds that the condition (a) holds, i.e. that

$$X_{\mathsf{A}_{\mathscr{A}'}} \subseteq X_{Env_{\mathcal{E}}(\mathbb{E})} \ .$$

Generally, this first property also implies that

$$\forall (X_{Env_{\mathcal{E}}(\mathbb{E})}, \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})}) \text{ where } \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}} \text{ and } \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G} \neq \emptyset : \quad (22)$$
$$\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}'} \subseteq \mathsf{A}_{\mathscr{A}} \ .$$

As a second property, consider that G' is the union of each intersection $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B}$ where (X, B) is an element where $\emptyset \neq \mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ and $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$. Given the fact that $X_{\mathsf{A}_{\mathscr{A}'}} \subseteq X_{Env_{\mathcal{E}}(\mathbb{E})}$ and in accordance with Lemma 1, Proposition 5,

and Definition 2, it follows that the condition (b) holds through the relation , i.e. it holds that

$$X_{\mathsf{G}'} \subseteq X_{Env_{\mathcal{E}}(\mathbb{E})} \cup X \ .$$

Generally, this also implies that

$$\forall (X, \mathsf{B}) \text{ where } \mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G} \text{ and } \mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B} : \ \mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}' \subseteq \mathsf{G} \ . \quad (23)$$

Now, for the if case, assume that the conditions (i) and (ii) hold with respect to $(\mathscr{A}', \mathsf{G}', X)$ for an arbitrary set of elements $\mathcal{E}$ containing an element (X, B) and where the pair $(X_{Env_{\mathcal{E}}(\mathbb{E})}, \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})})$ is the environment of (X, B). The condition (ii) and the relation (22) imply that

$$\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \subseteq \mathsf{A}_{\mathscr{A}'} \subseteq \mathsf{A}_{\mathscr{A}} \ . \quad (24)$$

The condition (i) and the relation (23) imply that

$$\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}' \subseteq \mathsf{G} \ .$$

This and the relation (24) imply that

$$\mathsf{A}_{\mathscr{A}'} \cap \mathsf{B} \subseteq \mathsf{G}' \subseteq \mathsf{G} \ . \quad (25)$$

Furthermore, in accordance with Theorem 1, the conditions (i) and (ii) imply that $\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{B} \neq \emptyset$. This and the relations (24) and (25) imply that

$$\mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})} \cap \mathsf{G}' \neq \emptyset \ . \quad (26)$$

Finally, the relation $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$ of the condition (i) and the fact that $\mathsf{A}_{\mathscr{A}'} \subseteq \mathsf{A}_{\mathscr{A}}$ and $\mathsf{G}' \subseteq \mathsf{G}$, as expressed in the relation (24) and (25), respectively, imply that

$$\mathsf{A}_{\mathscr{A}'} \cap \mathsf{G}' \subseteq \mathsf{B} \ . \quad (27)$$

The relations (24)-(27) imply that the conditions (i') and (ii') hold.

For the if-only case, assume that the conditions (i') and (ii') hold for an arbitrary set of elements $\mathcal{E}$ containing an element (X, B) and where the pair $(X_{Env_{\mathcal{E}}(\mathbb{E})}, \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})})$ is the environment of (X, B). From the first and second property of the contract $(\mathscr{A}', \mathsf{G}', X)$, it directly follows that (X, B) and $(X_{Env_{\mathcal{E}}(\mathbb{E})}, \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})})$ are such the condition (i) and (ii) hold, respectively.

Thus, given the contract $(\mathscr{A}, \mathsf{G}, X)$ and the set of variables $X_{Env_{\mathcal{E}}(\mathbb{E})}$, it can be concluded that there exists a contract $(\mathscr{A}', \mathsf{G}', X)$ where the conditions (a) and (b) hold such that for each set of elements containing an

element (X, B) and where the pair $(X_{Env_{\mathcal{E}}(\mathbb{E})}, \mathsf{B}_{Env_{\mathcal{E}}(\mathbb{E})})$ is the environment of (X, B), it holds that the conditions (i') and (ii') hold if and only if the conditions (i) and (ii) hold. □

Theorem 5. *Given a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and a set of contracts $\{\mathcal{C}_i\}_{i=1}^N$ where $\mathcal{C}_i = (\mathscr{A}_i, \mathsf{G}_i, X_i)$ and $X \subseteq \bigcup_{i=1}^N X_i$, the two-level contract hierarchy $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$ is proper if there exists a proper composition structure of $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$.*

Proof. Given a contract $\mathcal{C} = (\mathscr{A}, \mathsf{G}, X)$ and a set of contracts $\{\mathcal{C}_i\}_{i=1}^N$ where $\mathcal{C}_i = (\mathscr{A}_i, \mathsf{G}_i, X_i)$ and $X \subseteq \bigcup_{i=1}^N X_i$, assume that there exists a proper composition structure of the contract hierarchy $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$. In accordance with Definition 8, assume that there exists an arbitrary set of elements $\{(X_i, \mathsf{B}_i)\}_{i=1}^N$ such that $\mathsf{A}_{\mathscr{A}_i} \cap \mathsf{B}_i \subseteq \mathsf{G}_i$ and $\mathsf{A}_{\mathscr{A}_i} \cap \mathsf{G}_i \subseteq \mathsf{B}_i$ holds for each i.

With the intent to first show that the composition (X, B) of $\{(X_i, \mathsf{B}_i)\}_{i=1}^N$ onto X is such that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ holds, consider the fact that $\mathsf{A}_{\mathscr{A}_i} \cap \mathsf{B}_i \subseteq \mathsf{G}_i$ holds for each i. In accordance with Definition 9, this and since each assertion that is a direct predecessor of an assumption in $\mathscr{A}_i$ is either a guarantee G_j or an assumption in $\mathscr{A}$, it follows that $\mathsf{A}_{\mathscr{A}} \cap \bigcap_{i=1}^N \mathsf{B}_i \subseteq \bigcap_{i=1}^N \mathsf{G}_i$ in accordance with the condition (ii) of Definition 10. This and condition (i) of Definition 10 imply that $\mathsf{A}_{\mathscr{A}} \cap \bigcap_{i=1}^N \mathsf{B}_i \subseteq \mathsf{G}$. In accordance with Section 2.1 and Proposition 1, this and considering that condition (iii) implies that neither $\mathsf{A}_{\mathscr{A}}$ nor G constrains any subset of $\bigcup_{i=1}^N X_i \setminus X$, it follows that $\mathsf{A}_{\mathscr{A}} \cap \widehat{proj}_X(\bigcap_{i=1}^N \mathsf{B}_i) \subseteq \mathsf{G}$ due to the fact that $X = \bigcup_{i=1}^N X_i \setminus (\bigcup_{i=1}^N X_i \setminus X)$. In accordance with Definition 3, this means that it holds that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$.

With the intent to now show that the composition (X, B) of $\{(X_i, \mathsf{B}_i)\}_{i=1}^N$ onto X also is such that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$ holds, consider the fact that $\mathsf{A}_{\mathscr{A}_i} \cap \mathsf{G}_i \subseteq \mathsf{B}_i$ holds for each i. In accordance with Definition 9, this and since each assertion that is a direct predecessor of an assumption in $\mathscr{A}_i$ is either a guarantee G_i or an assumption in $\mathscr{A}$, it trivially follows that $\mathsf{A}_{\mathscr{A}} \cap \bigcap_{i=1}^N \mathsf{G}_i \subseteq \bigcap_{i=1}^N \mathsf{B}_i$ in accordance with the condition (ii) of Definition 10. This and given that it holds that $\bigcap_{i=1}^N \mathsf{B}_i \subseteq \widehat{proj}_X(\bigcap_{i=1}^N \mathsf{B}_i)$ in accordance with the relations (1) and (2), it follows that $\mathsf{A}_{\mathscr{A}} \cap \bigcap_{i=1}^N \mathsf{G}_i \subseteq \mathsf{B}$ in accordance with Definition 3. In accordance with the Section 2.1 and Proposition 1, this and due to the fact that Definition 3 and the condition (iii) of Definition 10 imply that neither $\mathsf{A}_{\mathscr{A}}$ nor B can constrain any non-empty subset of $\bigcup_{i=1}^N X_i \setminus X$, it holds that $\mathsf{A}_{\mathscr{A}} \cap \widehat{proj}_{\Xi \setminus (\bigcup_{i=1}^N X_i \setminus X)}(\bigcap_{i=1}^N \mathsf{G}_i) \subseteq \mathsf{B}$. This and the condition (iv) of Definition 10 imply that $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$.

Since the set of elements $\{(X_i, \mathsf{B}_i)\}_{i=1}^N$ was chosen arbitrarily, it holds that the composition of *each set of elements* $\{(X_i, \mathsf{B}_i)\}_{i=1}^N$ onto X is such that the relations $\mathsf{A}_{\mathscr{A}} \cap \mathsf{B} \subseteq \mathsf{G}$ and $\mathsf{A}_{\mathscr{A}} \cap \mathsf{G} \subseteq \mathsf{B}$ hold, if each element (X_i, B_i) is such that $\mathsf{A}_{\mathscr{A}_i} \cap \mathsf{B}_i \subseteq \mathsf{G}_i$ and $\mathsf{A}_{\mathscr{A}_i} \cap \mathsf{G}_i \subseteq \mathsf{B}_i$ hold. This means that $(\mathcal{C}, \{\mathcal{C}_i\}_{i=1}^N)$ is proper in accordance with Definition 8, which completes the proof. □

Paper B

Extending Contract Theory with Safety Integrity Levels

Jonas Westman • Mattias Nyberg

In *High Assurance Systems Engineering (HASE)*, 2015 IEEE 16th International Symposium on.

Abstract

In functional safety standards such as ISO 26262 and IEC 61508, Safety Integrity Levels (SILs) are assigned to top-level safety requirements on a system. The SILs are then either *inherited* or *decomposed* down to safety requirements on sub-systems, such that 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. Present contract theory has previously been shown to provide a suitable foundation to structure safety requirements, but does not include support for the use of SILs. An extension of contract theory with the notion of SILs is therefore presented. As a basis for structuring the breakdown of safety requirements, a graph, called a *contract structure*, is introduced that provides a necessary foundation to capture the notions of SIL inheritance and decomposition in the context of contract theory.

1 Introduction

The concept of Safety Integrity Levels (SILs) is central in various safety standards, e.g. [1–4], and can be described as a measure of the required reliability of a system or component, in order to achieve a tolerable level of risk. This means that, the higher the SIL, the greater the amount of *safety measures*, i.e. activities or technical solutions that increase the reliability of a system, needs to be applied, in order to mitigate the occurrence of dangerous system failures.

Typically, e.g. in [1, 2], SILs are assigned to top-level safety requirements. 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.

Consider that a requirement for a sub-system is assigned with a lower SIL than what is actually needed for the system to be sufficiently reliable in fulfilling a top-level safety requirement. This means that even if the sub-system is sufficiently reliable in fulfilling its requirement, it cannot be

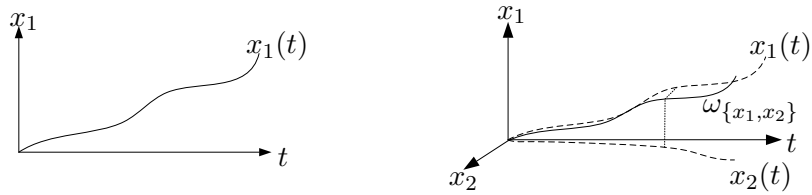
guaranteed that a tolerable level of risk is met. It is therefore imperative that the assignment of SILs to the requirements for a system is done in a correct manner.

In [5, 6], the contract theory [7, 8] of SPEEDS [9], targeted for the design of Cyber-Physical Systems (CPS), is extended to allow contracts that are not limited to the interface of a system or component. As further shown in [5], the extended theory of contracts provides a suitable foundation to structure safety requirements. The works in [5–8], do not, however, discuss the relation between contracts and SILs, and do not, therefore, provide the necessary support to argue over the fact that a breakdown of safety requirements for a system has been performed correctly with respect to achieving a tolerable level of risk.

The main contribution of the present paper is an extension of contract theory [5–8] with the notion of SILs. This includes definitions that specifies exact conditions under which SIL inheritance and decomposition can be applied when using contracts to structure safety requirements. As a basis for structuring the breakdown of safety requirements and the individual tracing of safety requirements at a lower level back to their top-level safety requirement, a graph, called a *contract structure*, is introduced.

Out of an extensive literature study (See Sec. 2.3) of contract approaches [7, 8, 10–24], only [10] was found to discuss the relation between SILs and contracts in any depth. In [10], inspired by the work in [25], *'safety contracts'* are used as a mean to explicate the relation between the integrity levels of assumptions on the inputs and the guarantee on the outputs of a software component. The notion of inheritance of integrity levels inbetween requirement breakdown levels is also discussed, but the discussion is kept at non-formal level and is limited to the software domain only. In contrast to [10], the present paper captures the relations between SILs and contracts in a formal context and, in addition, covers the area of Cyber Physical Systems (CPS) in general; not only software.

In Sec. 2, the theory of contracts [5–8] is presented. In Sec. 3, *contract structures* are introduced in order to capture the breakdown of safety requirements. In Sec. 4, SILs are described from the perspectives of safety standards. In Sec. 5, present contract theory is extended with SILs. In Sec. 6, the paper is summarized and conclusions are drawn.



(a) (b)

Figure 1. In (a), a trajectory of values of x_1 is shown. In (b), a run $\omega_{\{x_1,x_2\}}$ is shown, consisting of two pairs containing the trajectory shown in (a) and another trajectory of values of x_2.

2 Present Contract Theory

In this section, the essence of the theory of contracts as described in [5–8] is presented. The concepts are later used in Sec. 3 and 5 where contracts are structured for an architecture of elements and related to the notion of SILs, respectively.

2.1 Assertions and Runs

Let $X = \{x_1, \ldots, x_N\}$ be a set of variables. Consider a trajectory of values of the variable x_1 over a time window starting at a certain time t_0, e.g. as shown in Fig. 1a. A tuple of such trajectories, one for each variable in X, sorted according to a global ordering with respect to the identifiers of the variables, is called a *run* for X, denoted ω_X. For example, a run $\omega_{\{x_1,x_2\}}$ is shown in Fig. 1b as a solid line, consisting of the trajectory shown in Fig. 1a and another trajectory of values of x_2, both represented as dashed lines.

Given a set of variables X', an *assertion* W over X' is a non-empty set of runs for X'. Note that rather than specifying an assertion by explicitly declaring its set of runs, assertions can be specified by a set of constraints, e.g. by equations, inequalities, or logical formulas.

Given an assertion W over $X = \{x_1, \ldots, x_N\}$, and another set of variables $X' \subseteq X$, the *projection* [5, 7, 22, 24] of W onto X', written $proj_{X'}(\mathsf{W})$, is the set of runs obtained when the trajectory of values of each variable $x_i \notin X'$ is removed from each run in W. Using notation of relational al-

gebra [26], it holds that $proj_{X'}(\mathsf{W}) = \pi_{X'}(\mathsf{W})$. For example, consider an assertion $\{\omega_{\{x_1,x_2\}}\}$ where $\omega_{\{x_1,x_2\}}$ is the run shown in Fig. 1b. The projection of $\{\omega_{\{x_1,x_2\}}\}$ onto $\{x_1\}$ is the run containing only the trajectory of values of the variable x_1 as shown in Fig. 1a.

In the following, for the sake of readability, it is assumed that prior to using set operations (e.g. $\cap$, $\cup$, and *proj*) and set relations (e.g. $\subseteq$) on assertions over dissimilar sets of variables, the assertions are first extended to the union of the sets of variables using the operation of inverse projection as described in [5, 6]. For example, given two assertions W and W' over the set of variables X and X', respectively, the shorthand notation $\mathsf{W} \cap \mathsf{W}'$ is assumed to mean that both W and W' are first extended to the set $X \cup X'$, prior to performing the intersection.

2.2 Elements and Architectures

This section starts by introducing the concept of *elements*[1], in order to model any entity of a CPS, such as software, hardware, or physical entities.

Definition 1 (Element)**.** An *element* $\mathbb{E}$ is an ordered pair (X, B) where:

a) X is a set of variables, called the *interface* of $\mathbb{E}$ and where each $x \in X$ is called a *port variable*; and

b) B is an assertion over X, called the *behavior* of $\mathbb{E}$.

Port variables model tangible quantities of the element from the perspective of an external observer to the element, and the behavior models the static and dynamic constraints that the element imposes on the port variables, independent of its surroundings. For example, consider a potentiometer $\mathbb{E}_{pot} = (X_{pot}, \mathsf{B}_{pot})$ where $X_{pot} = \{v_{ref}, v_{branch}, v_{gnd}\}$. The port variables v_{ref}, v_{branch}, and v_{gnd} model the reference, branch, and ground voltages, respectively. Furthermore, h models the position $(0 - 100\%)$ of the 'slider' that moves over the resistor and branches the circuit. Given a simplified model where currents are neglected, the behavior B_{pot} can be specified by the equation $h = \frac{v_{branch} - v_{gnd}}{v_{ref} - v_{gnd}}$.

A set of elements can be structured in order to model a CPS, its parts, and its surroundings. Similar to e.g. [2, 27], such a structure will be referred

[1]The concept corresponds to a component in [7, 8] and is, essentially, in accordance with the concept of an element in ISO 26262 [2].

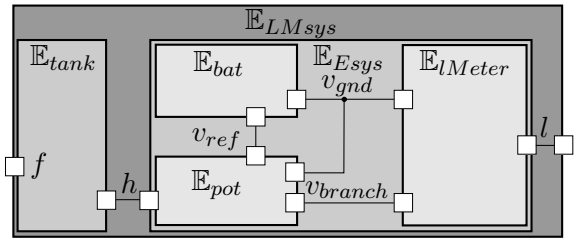


Figure 2. An architecture $\mathscr{A}_{LMsys}$ of a "Level Meter system".

to as an *architecture*, which, in the present paper, will be denoted with the symbol $\mathscr{A}$. Prior to presenting the formal definition of an architecture, the concept is introduced informally by describing an architecture $\mathscr{A}_{LMsys}$ of a "Level Meter system" (LM-system) $\mathbb{E}_{LMsys}$, as shown in Fig. 2 where a set of elements, each represented as a rectangle filled with gray with white boxes on its edges that symbolize its port variables, is structured heirarchically.

As shown in Fig. 2, the LM-system $\mathbb{E}_{LMsys}$ consists of a tank $\mathbb{E}_{tank}$ and an electric-system $\mathbb{E}_{E-sys}$. The electric-system $\mathbb{E}_{E-sys}$ consists of the potentiometer $\mathbb{E}_{pot}$ as previously described, a battery $\mathbb{E}_{bat}$ and a level meter $\mathbb{E}_{lMeter}$ where the behaviors B_{bat} and B_{lMeter} of $\mathbb{E}_{bat}$ and $\mathbb{E}_{lMeter}$ are specified by the equations $v_{ref} - v_{gnd} = 5V$ and $l = \frac{v_{branch} - v_{gnd}}{5}$, respectively. The slider h 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.

Notably, since each part of the electric-system $\mathbb{E}_{Esys}$ will have quantities that may not be perceivable when the parts are integrated with each other, e.g. v_{ref}, a port variable x of a child of $\mathbb{E}_{Esys}$ where $x \notin X_{Esys}$ cannot be a member of an interface of a non-descendant of $\mathbb{E}_{Esys}$, e.g. $\mathbb{E}_{tank}$. In order to further relate the individual behaviors of the children of $\mathbb{E}_{Esys}$ with the behavior of $\mathbb{E}_{Esys}$, the individual behaviors are first combined with each other using the intersection operator and subsequently restricted to the interface of $\mathbb{E}_{Esys}$ using the projection operator, in accordance with the Sec. 2.1, i.e., $\mathsf{B}_{Esys} = proj_{X_{Esys}}(\mathsf{B}_{pot} \cap \mathsf{B}_{bat} \cap \mathsf{B}_{lMeter})$.

The formal definition of an architecture now follows:

Definition 2 (Architecture)**.** An *architecture* $\mathscr{A}$ 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 $\mathsf{B} = proj_X(\bigcap_{i=1}^{N} \mathsf{B}_i)$; and

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

2.3 Contracts

The notion of *Contracts* was first introduced in [12] as a pair of pre- and post-conditions, to be used as a specification in object-oriented programming. The principles behind contracts can, however, be traced back to early ideas on proof-methods [28–30] and compositional reasoning/verification [13–15]. Since then, several frameworks for compositional reasoning [16, 17] have emerged and the work in [12] has been extended to e.g. component-based design [18], analog systems [19], modal specifications [20], and probabilistic systems [21]. Recently, a theory on contracts for CPS was presented in [7, 8]. Recent works also include abstract formalizations of contract theory [22–24] and an extension [5, 6], that allows assumptions and guarantees that are not limited to the interface of elements.

Definition 3 (Contract)**.** A *contract* $\mathcal{C}$ is a pair $(\mathcal{A}, \mathsf{G})$, where G is an assertion, called a *guarantee*, and $\mathcal{A}$ is a set of assertions $\{\mathsf{A}_i\}_{i=1}^{N}$ where each A_i is called an *assumption of* G.

A contract $\mathcal{C} = (\mathcal{A}, \mathsf{G})$ is intended to be used as a specification for an element $\mathbb{E} = (X, \mathsf{B})$ where the guarantee G expresses an intended property under the responsibility of $\mathbb{E}$, given that any environment of the element fulfills the assumptions in $\mathcal{A}$. The element $\mathbb{E}$ *satisfies* [5–8, 22–24] the contract $\mathcal{C}$ if the behavior B and the assumptions of G together fulfill G, i.e. if the relation $(\bigcap_{i=1}^{N} \mathsf{A}_i) \cap \mathsf{B} \subseteq \mathsf{R}$ holds.

As an illustrative example, consider the architecture $\mathscr{A}_{LMsys}$ as shown in Fig. 2 and a contract $\mathcal{C}_{lMeter} = (\{\mathsf{A}_{lMeter}\}, \mathsf{G}_{lMeter})$ for the level meter $\mathbb{E}_{lMeter}$ as shown in Fig. 3 where the dashed lines represent the variables that A_{lMeter} and G_{lMeter} are expressed over, respectively. The assertion G_{lMeter}, specified by the equation $l = f$, expresses that the responsibility of $\mathbb{E}_{lMeter}$ is to guarantee that the presented fuel level l, shown by the meter, shall correspond to the level f in the tank. However, in order for the level meter $\mathbb{E}_{lMeter}$ to be able to ensure that G_{lMeter} holds, the voltage measured between v_{branch} and v_{gnd} on $\mathbb{E}_{lMeter}$ must map to a specific level in the

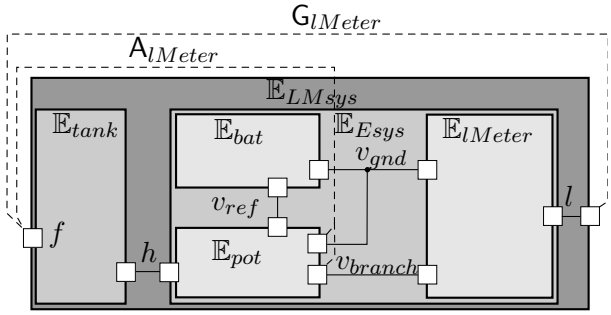


Figure 3. A contract $\mathcal{C}_{lMeter}$ for the element $\mathbb{E}_{lMeter}$.

tank. That is, the assumption A_{lMeter} of G_{lMeter} is specified by the equation $f = \frac{v_{branch} - v_{gnd}}{5}$.

Formulated differently, the guarantee G_{lMeter} expresses a *requirement on* $\mathbb{E}_{lMeter}$, given an environment that fulfills the assumptions. Considering that the overall aim is to relate the concept of SILs in safety standards to the notion of contracts, and that safety standards rely on the use of the term *'requirements'* rather than *'guarantees'*, a guarantee G, will, in the following instead be referred to as a requirement R.

3 Contract Structures

Consider that it is infeasible to verify that the root element $\mathbb{E}_r$ of an architecture satisfies a contract $\mathcal{C}_r$, due to the complexity of $\mathbb{E}_r$. A solution to such an issue, is to iteratively split each contract $\mathcal{C}$ for an element $\mathbb{E}$ into a set of contracts $\{\mathcal{C}_i\}_{i=1}^{N}$ for each child $\mathbb{E}_i$ of $\mathbb{E}$ with the intent that:

$$\text{each leaf element } \mathbb{E}_j \text{ satisfies } \mathcal{C}_j \implies \mathbb{E}_r \text{ satisfies } \mathcal{C}_r. \tag{1}$$

If an architecture only consists of two hierarchical level, then property (1) corresponds, in essence, to the notion of *completeness* in ISO 26262 [2], *dominance/refinement* in [5–8, 23, 24], and the *composition principle* in [16].

This section introduces the concept of a graph, called a *contract structure* $\mathfrak{C}$ that organizes the breakdown of requirements on the elements of an architecture with the intent of having the property as expressed in the property (1). Contract structures are further used in Sec. 5.1 and 5.2 in order to capture the notion SIL inheritance and decomposition, respectively. Prior

to presenting the formal definition of a contract structure in Sec. 3.3, an underlying concept of using assumptions as explicit requirements is described in Sec. 3.1.

3.1 Using Assumptions as Explicit Requirements

Consider a scenario where the environment of the element $\mathbb{E}_{lMeter}$ is unknown, e.g. when developing $\mathbb{E}_{lMeter}$ *"out-of-context"* [2]. The assumption A_{lMeter} of the *requirement* G_{lMeter}, as shown in Fig. 3, hence expresses the conditions *any environment* needs to fulfill, in order for the behavior of $\mathbb{E}_{lMeter}$ to guarantee that G_{lMeter} holds. If the environment is known, however, such as in the context of the architecture $\mathscr{A}_{LMsys}$, the assumption A_{lMeter} of G_{lMeter} rather expresses as a requirement on the potentiometer $\mathbb{E}_{pot}$. This was also observed in [5, 31] where, in the context of an architecture, assumptions are in fact references to other requirements.

Therefore, in the context of a specific architecture, it is not needed to use explicit assumptions separated from requirements, and an assumption of a requirement on an element $\mathbb{E}$, will, in the following, correspond to a *requirement on an element in the environment of* $\mathbb{E}$. The use of explicit assumptions is, however, relevant for out-of-context development and is discussed in Remark 1.

3.2 Contract Structure for the Level Meter-system

Consider that each requirement R_i of a contract $\mathcal{C}_i = (\mathcal{A}_i, \mathsf{R}_i)$ for an element $\mathbb{E}_i \in \mathscr{A}_{LMsys}$ is represented by a node in an edge-labeled directed graph as an overlay onto the hierarchical structure of the elements of $\mathscr{A}_{LMsys}$ as shown in Fig. 4. A requirement R_i is an assumption of another requirement R_j, if there exists an arc labeled "Assumption of" from R_i to R_j, visualised 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 the requirement R_{lMeter}.

As also shown in Fig. 4, an incoming arc labeled "Fulfills", visualized as an arrow, to a requirement R_i on an element $\mathbb{E}_i$ from a requirement R_j on a child $\mathbb{E}_j$ of $\mathbb{E}_i$, represents the intent of R_j fulfilling R_i. For example, the arc from the requirement R_{E-sys} to R_{LMsys} represents the intent of $\mathsf{R}_{E-sys} \subseteq \mathsf{R}_{LMsys}$.

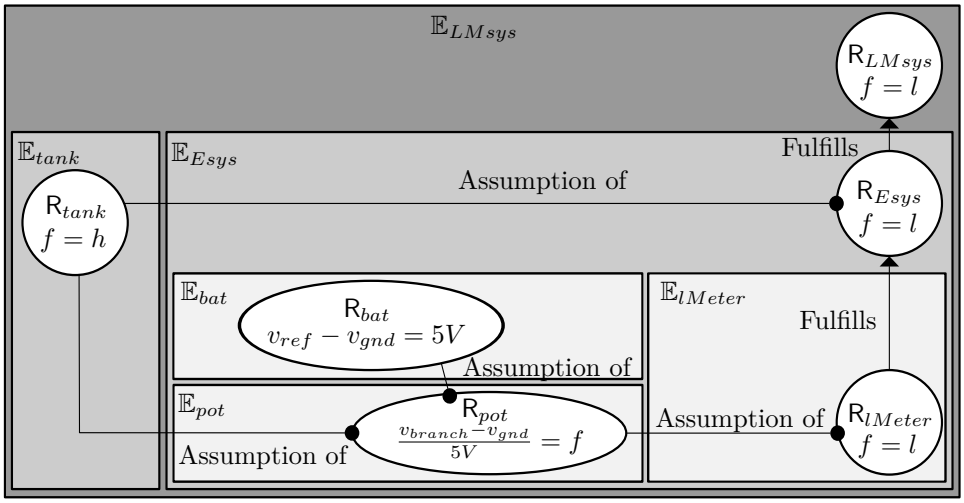


Figure 4. A contract structure $\mathfrak{C}_{LMsys}$ for the architecture $\mathscr{A}_{LMsys}$.

3.3 Contract Structure

The formal definition of a contract structure follows:

Definition 4 (Contract Structure)**.** Given an architecture $\mathscr{A}$, and a set of contracts $\{(\mathcal{A}_{i,j}, \mathsf{R}_{i,j})\}_{j=1}^{N_i}$ for each element $\mathbb{E}_i$ of $\mathscr{A}$, where each assumption in each set $\mathcal{A}_{i,j}$ is either:

a) a requirement on a sibling of $\mathbb{E}_i$; or

b) an assumption of a requirement $\mathsf{R}_{k\neq i,l}$ on the parent of $\mathbb{E}_i$,

then a *contract structure* $\mathfrak{C}$ is an arc-labeled Directed Acyclic Graph (DAG), such that:

i) the requirements $\mathsf{R}_{i,j}$ are the nodes in $\mathfrak{C}$;

ii) each arc is uniquely labeled either "Assumption of" or "Fulfills";

iii) there is an arc labeled "Assumption of" from $\mathsf{R}_{i,j}$ to $\mathsf{R}_{k\neq i,l}$, if and only if $\mathsf{R}_{i,j}$ is an assumption of $\mathsf{R}_{k\neq i,l}$;

iv) if there is an arc labeled "Fulfills" from $\mathsf{R}_{i,j}$ to $\mathsf{R}_{k\neq i,l}$, then $\mathsf{R}_{k\neq i,l}$ is a requirement on the parent of $\mathbb{E}_i$;

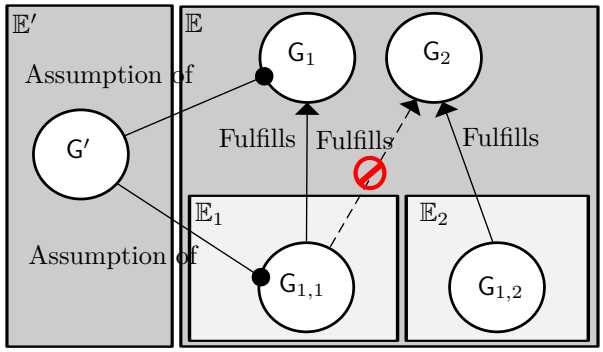


Figure 5. A contract structure that would not be a contract structure if the dashed arc is added to the graph.

v) if $\mathsf{R}_{i,j}$ is reachable from an assumption $\mathsf{R}_{k\neq i,l}$ of a requirement on the parent of $\mathbb{E}_i$, then $\mathsf{R}_{k\neq i,l}$ is also an assumption of any requirement $\mathsf{R}_{m\neq k,n}$ on the parent of $\mathbb{E}_i$ and $\mathsf{R}_{m\neq k,n}$ is reachable from $\mathsf{R}_{i,j}$;

vi) each requirement on a non-leaf element $\mathbb{E}$ has a predecessor that is a requirement on a child of $\mathbb{E}$.

As discussed in Sec. 3.1 and as also shown in Fig. 1, the conditions a) and b) of Def. 4 express that an assumption of a requirement on an element $\mathbb{E}$ always correspond to a *requirement on an element in the environment of* $\mathbb{E}$, i.e. an assumption is either a requirement on a sibling of $\mathbb{E}$, or an assumption of a requirement on the parent of $\mathbb{E}$.

Consider the contract structure shown in Fig. 5 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 (v) of Def. 4. In Fig. 5, the intent is that *the behavior* B *of* $\mathbb{E}$ *is to fulfill* R_2*, without relying on any assumptions concerning the environment of* $\mathbb{E}$. However, if the dashed arc is added, then the above-mentioned statement is contradicted, since it would mean that in order for B to fulfill R_2, the behavior of the sibling $\mathbb{E}'$ of $\mathbb{E}$ needs to fulfill R'.

The condition (vi) of Def. 4 expresses that the requirement on an element $\mathbb{E}$ must be linked to a requirement on a child of $\mathbb{E}$.

Contract structures will be used to capture the notion of SIL inheritance and decomposition in the context of contracts in Sec. 5, which also presents a theorem that shows that a contract structure organizes the breakdown of

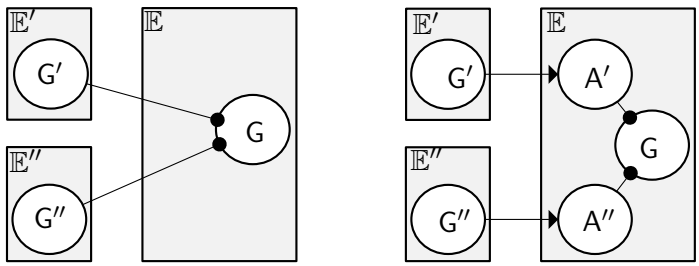


Figure 6. A possible extension to Def. 4 is shown in (b) where, in contrast to (a), assumptions are used as intermediate nodes.

the requirements on the elements of an architecture to achieve the property as expressed in (1). Prior to that, however, an overview of the use of SILs in different safety standards is presented in the following section.

Remark 1 (Extension for Out-of-Context Development). A trivial extension to Def. 4 is to allow the use of intermediate assumption nodes, e.g. as shown in Fig. 6b, which can be compared to Fig. 6a where the requirements R' and R'' are assumptions to R. Such an extension is useful for e.g. out-of-context development as discussed in Sec. 3.1.

4 Safety Integrity Levels in Safety Standards

The concept of SILs is central in various safety standards, e.g. IEC 61508 [1] - a generic standard for functional safety, EN 50128 [3] for the railway industry, ISO 26262 [2] for automotive, and Def Stan 00-56 [4] for the defense industry. Although the aim is to consider SILs from a general standpoint, due to space restrictions, this paper is focused to SILs from the perspective of IEC 61508 and ISO 26262, as presented in Sec. 4.1 and 4.2. The reason for choosing IEC 61508 is that it is a generic standard, applicable to any safety-related system that incorporates Electric/Electronic/Programmable Electronic (E/E/PE) parts, and also due to the fact that one of its major objectives is to serve as an archetype for other more domain-specific standards. The reason for choosing ISO 26262 is that it is one of the most recent functional safety standards and it has a high focus on requirements engineering with a notion of SIL inheritance and decomposition more explicit than

in [1, 3, 4].

4.1 SILs in IEC 61508

In order to evaluate the risk, i.e. the combination of the probability of harm and the severity of such a harm, of deploying an *Equipment Under Control* (EUC), i.e. *"equipment, machinery, apparatus or plant used for manufacturing, process, transportation, medical or other activities"* [1], the EUC and its environment is analyzed in order to identify hazardous events, i.e. events that may result in harm. The risks of the hazardous events are then assessed without considering mechanisms of the EUC that might lower such a risk.

A *safety function* is determined for each hazardous event, formulated with the intent to prevent the hazardous events from occurring. Safety functions are then either allocated to *E/E/PE safety-related systems* of the EUC or to *other risk reduction measures*. With the intent to achieve a tolerable level of risk, elements of E/E/PE safety-related systems are determined a *SIL*:

> *"a discrete level (one out of a possible four), corresponding to a range of probabilities of an E/E/PE safety-related system satisfactorily performing the specified safety functions under all the stated conditions within a stated period of time, where safety integrity level 4 has the highest level of safety integrity and safety integrity level 1 has the lowest"* [1].

4.2 ASILs in ISO 26262

A development according to ISO 26262 revolves around an *item*, i.e. *"a system that implements a function at a vehicle level"* [2]. Similar to IEC 61508, the risks of the different hazardous events that are relevant for the item are assessed without considering mechanisms that might lower such a risk. Based on the risk assessment, each hazardous event is then determined an *Automotive Safety Integrity Level* (ASIL), i.e.

> *"one of four levels to specify the item's or element's [an item consists of elements] necessary requirements of ISO 26262 and safety measures to apply for avoiding an unreasonable residual*

> *risk* [i.e. risk that remains after the deployment of safety measures] *with D representing the most stringent and A the least stringent level"* [2].

A top-level safety requirement, i.e. a *safety goal* is determined for each hazardous event, formulated with the intent to prevent the hazardous event from occurring. As a consequence, the safety goal is assigned the ASIL determined for the hazardous event.

4.2.1 ASIL Inheritance

In ISO 26262, safety requirements are structured at different hierarchical levels where the intent is that *"safety requirements at one level fully implement all safety requirements of the previous level"* [2]. For example, if two so called Functional Safety Requirements (FSRs) are *derived* from a safety goal, then the intent is that the FSRs shall *implement* the safety goal. If the safety goal has been assigned an ASIL D, for example, then the two FSRs will *inherit* SIL D. If one of the FSRs is also derived from another safety goal, then that FSR will inherit the highest ASIL of the safety goals.

4.2.2 ASIL Decomposition

Consider a case where either one of the FSRs can, in fact, implement the safety goal alone and that the FSRs are allocated to two elements that are *'sufficiently independent'*. Given such a case, it is possible to assign lower SILs to the FSRs (i.e. in this case an ASIL B and C, for example) than to the safety goal by applying *'ASIL 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"* [2].

5 Safety Integrity Levels for Contracts

In this section, the contract theory as described in Sec. 2 is extended with the notion of SILs, as described in Sec. 4. This includes formal definitions of both SIL inheritance and decomposition by relying on the concept of contract structures, as presented in Sec. 3.

Even though the intent is that the definitions in this section shall be in accordance with both ISO 26262 and IEC 61508, an interpretation is necessary both due to the fact that the standards deviate slightly in their definition of a SIL and that the definitions need to be tailored in order to fit with the other concepts presented in the paper.

Prior to presenting a formal definition of a SIL in the context of contract 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 $(\mathcal{A}, \mathsf{R})$, the *requirement* R *is violated* if a run ω of $\bigcap_{i=1}^{N} \mathsf{B}_i$ is executed where $\{\omega\} \not\subseteq \mathsf{R}$.

Definition 5 (Safety Integrity Level For a Requirement)**.** Given an architecture $\mathscr{A}$, a *SIL for a requirement* R, denoted SIL_{R}, is a uniquely assigned discrete level that corresponds to a target range of the probability that the requirement R is violated, during an arbitrary time interval of a predefined length.

Def. 5 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 ASIL for a requirement in ISO 26262 can be mapped to a range of the probability that the requirement is violated.

Consider that a SIL is assigned to the requirement R_{LMsys}, as shown in Fig 4, 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}.

5.1 SIL Inheritance in a Contract Structure

Prior to presenting a formal definition of SIL inheritance, the concept is introduced informally by the use of the following representative examples (a) and (b), as shown in Fig. 7:

a) In Fig. 7a, 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 the assumption R of R_1 and R_2. The highest SIL for R_1 and R_2, i.e. 2, is hence assigned to R;

b) In Fig. 7b, the intent is that the requirement R on the child $\mathbb{E}$ of an element $\mathbb{E}'$ is to fulfill both a requirement R_1 and R_2 on $\mathbb{E}$. The highest

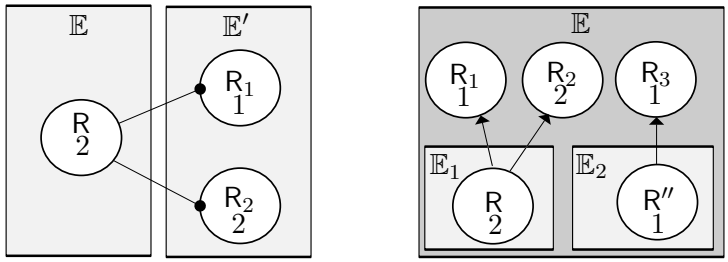


Figure 7. Two representative examples of SIL inheritance in a contract structure.

SIL for R_1 and R_2, i.e. 2, is hence assigned to R. The requirement R'' is intended to fulfill only R_3 and is hence assigned with the SIL 1.

As expressed in the two examples above, if the intent is to rely on that a requirement R is not violated in order to assure that a requirement R_i is not violated, then the requirement R should *inherit* the SIL of R_i. The behavior of $\mathbb{E}_{E-sys}$, for example, cannot fulfill the requirement R_{E-sys} unless the sensor is installed correctly in the tank, as expressed by R_{tank} and shown in Fig. 4. The SIL for R_{tank} is hence inherited from R_{E-sys}.

A formal definition of SIL inheritance will now follow below. Note that to fully understand the formal definition, SIL inheritance needs to be explained simultaneously with SIL decomposition. A detailed explanation will hence follow in Sec. 5.2. Part (a) and (b) of the definition corresponds to Fig 7a and 7b, respectively.

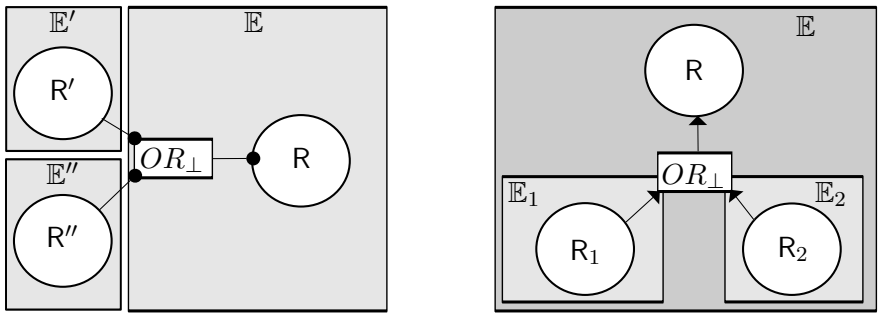
Definition 6 (SIL Inheritance in Contract Structure)**.** Given an architecture $\mathscr{A}$ and a contract structure $\mathfrak{C}$, *SIL inheritance* is the assignment of a SIL to a requirement R on an element $\mathbb{E}$ such that

$$SIL_{\mathsf{R}} = max(SIL_{\mathsf{R}_1}, \ldots, SIL_{\mathsf{R}_N})$$

where each R_i is either:

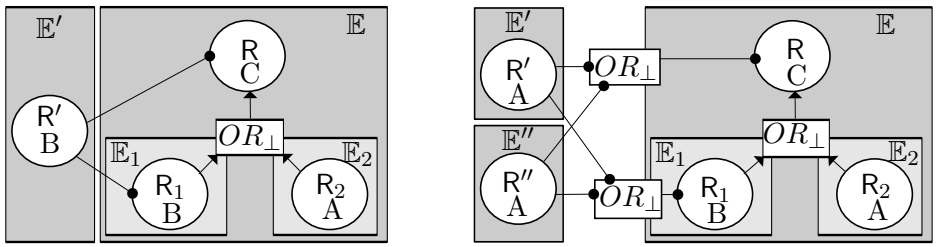
a) a requirement on a leaf element of $\mathscr{A}$ and R is both an assumption and a direct predecessor of R_i; or

b) a requirement on the parent of $\mathbb{E}$ and a direct successor of R.



(a) (b)

Figure 8. Two representative examples where the intent is to achieve redundancy.



(a) (b)

Figure 9. Examples where both SIL decomposition and inheritance is applied in contract structures.

5.2 SIL Decomposition in a Contract Structure

In this section, the concept of SIL decomposition in the context of a contract structure, is presented. As expressed in Sec. 4.2, 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 is extended with an '$OR_\perp$' node:

Definition 7 (Extension of Contract Structure)**.** i) each node is either a requirement $\mathsf{R}_{i,j}$ or an '$OR_\perp$' node and each requirement $\mathsf{R}_{i,j}$ is a node;

iii) If and only if $\mathsf{R}_{i,j}$ is an assumption of $\mathsf{R}_{k\neq i,l}$, then there exists an arc

labeled "assumption of" from $\mathsf{R}_{i,j}$ to either:

a) $\mathsf{R}_{k\neq i,l}$; or

b) an '$OR_{\perp}$' node that has exactly one outgoing arc to $\mathsf{R}_{k\neq i,l}$, labeled "assumption of";

vii) 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 requirement $\mathsf{R}_{k\neq i,l}$ on the parent of $\mathbb{E}_i$;

viii) each '$OR_{\perp}$' node has at least two incoming arcs and where any two incoming arcs to the '$OR_{\perp}$' node, are requirements on different elements.

The two following representative examples a) and b), also shown in Fig. 8a and 8b, capture two scenarios where the intent is to achieve redundancy:

a) As expressed in Fig. 8a, the intent is that it is sufficient that either one of the behaviors of two *"sufficiently independent"* [2] 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. 8b, 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 the 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 8 (SIL Decomposition in Contract Structure)**.** Given an architecture $\mathscr{A}$ and a contract structure $\mathfrak{C}$, *SIL Decomposition* is the assignment of a SIL to a requirement R of $\mathfrak{C}$ such that

$$\beta \leq SIL_{\mathsf{R}} \leq max(SIL_{\mathsf{R}_1}, \ldots, SIL_{\mathsf{R}_N}) \ ,$$

where each R_i is either:

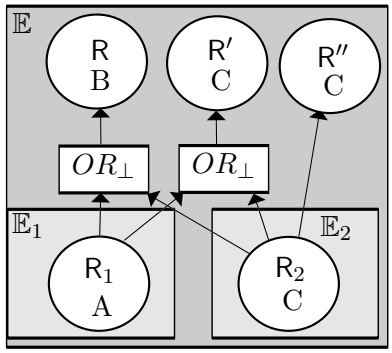


Figure 10. Example where both SIL decomposition and inheritance is applied in a contract structure.

a) a requirement on a leaf element of $\mathscr{A}$ and a direct successor of an $OR_\perp$ node where the $OR_\perp$ node has an incoming "assumption of" arc from R; or

b) a direct successor of an $OR_\perp$ where the $OR_\perp$ node has an incoming "fulfills" arc from R,

and where β is specified according to the context and the requirements of a given safety standard.

Consider Fig. 9a that is intended to clarify why inheritance of SILs, through the use of assumptions, only apply to leaf nodes, as expressed in (a) of Def. 6. 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 (b) of Def. 8 and ISO 26262. According to (a) of Def. 6, R′ should hence inherited the SIL B from the requirement R_1 on the leaf element $\mathbb{E}_1$, 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. 9b and expressed in (a) of Def. 8.

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. An example is shown in Fig. 10.

5.3 Verifying Contract Structures and Arguing for Safety

Now that formal definitions of SIL inheritance and decomposition has been presented, it will be shown that a contract structure organizes the breakdown of the requirements on the elements of an architecture in order to achieve the property as expressed in (1).

Theorem 1. *Given an architecture* $\mathscr{A}$*, and a set of contracts* $\{\mathcal{C}_{i,j}\}_{j=1}^{N_i}$ *for each element* $\mathbb{E}_i$ *of* $\mathscr{A}$ *and a contract structure* $\mathfrak{C}$ *where*

a) each requirement on the root element $\mathbb{E}_r$ *of* $\mathscr{A}$ *is over a subset of the interface of* $\mathbb{E}_r$*; and*

b) for each requirement R *with a set of incoming "fulfills' arcs from a set of requirements* $\{\mathsf{R}_1, \ldots, \mathsf{R}_N\}$*, the relation* $\bigcap_{i=1}^{N} \mathsf{R}_i \subseteq \mathsf{R}$ *holds,*

then the root element $\mathbb{E}_r$ *satisfies each contract* $\mathcal{C}_{r,k}$*, if each leaf element* $\mathbb{E}_i$ *satisfies each contract* $\mathcal{C}_{i,j}$*.*

The following illustrative example shows the use of Theorem 1.

Consider the contract structure in Fig. 11 where a redundant power supply $\mathbb{E}_{bat2}$ has been added to the architecture $\mathscr{A}_{lM-sys}$, as shown in Fig. 2 and 4. According to Def. 6.b), if R_{LMsys} has been assigned SIL C, SIL C is assigned to R_{E-sys} and, further also to R_{lMeter}. According to Def. 6.b), SIL C is assigned to R_{pot} and further also to R_{tank}. According to Def. 8.a) and the rules of decomposition in ISO 26262, SIL B and A is assigned to R_{bat} and R_{bat2}, respectively.

Since the relation $\mathsf{R}_{lMeter} \subseteq \mathsf{R}_{E-sys} \subseteq \mathsf{R}_{LMsys}$ holds, and that R_{LMsys} is over $\{l, f\} \subseteq X_{LMsys}$, it can be infered, through the use of Theorem 1, that if the leaf elements of $\mathscr{A}_{lM-sys}$ satisfy their contracts, then $\mathbb{E}_{lM-sys}$ satisfies $\mathcal{C}_{lM-sys}$. If the specific instructions of ISO 26262 are followed in order to ensure that each leaf requirement is not violated within the target probability as specified by its assigned SIL, then the argumentation can be made that the root requirement R_{LMsys} is not violated within the target probability as specified by its assigned SIL.

Remark 3. Since the requirements of a contract structure are organized as a Directed *Acyclic* Graph, the use of circular argumentation [16, 24] is avoided.

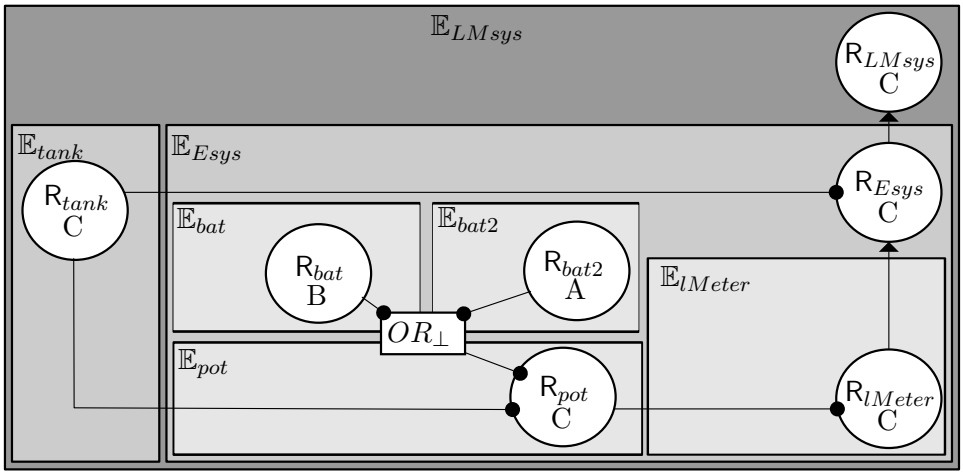


Figure 11. A contract structure for a modified version of the architecture $\mathscr{A}_{lM-sys}$ where redundancy has been introduced and SILs have been assigned to the requirements.

6 Conclusions

As mentioned in Sec. 1, although present contract theory [5–8] provides a suitable foundation to structure safety requirements, there is insufficient support for assigning SILs to requirements structured as contracts.

To address this, the notion of a contract structure was first introduced in Sec. 3 as a basis for structuring the breakdown of safety requirements and the individual tracing of safety requirements at a lower level back to their top-level safety requirement. Through the use of contract structures, contract theory [5–8] was extended with formal definitions of both SIL inheritance and decomposition in Sec. 5.1 and 5.2. In Sec. 5.3, Theorem 1 was introduced that ensures that a breakdown of safety requirements, structured as contracts, has been performed correctly. A correct breakdown of safety requirements is necessary in order to argue over the fact that a tolerable level of risk of a system has been met.

Considering the strict demands from recent functional safety standards [1, 2] on the structuring of safety requirements - including SILs, this paper offers a necessary extension of contract theory [5–8] that supports a systematic and correct way of assigning SILs for safety requirements structured as contracts.

References

[1] IEC 61508, "Functional safety of electrical/electronic/programmable electronic safety-related systems," 2010.

[2] ISO 26262, "Road vehicles-Functional safety," Geneva, Switzerland, 2011.

[3] EN 50128, "Railway applications - communication, signalling and processing systems - software for railway control and protection systems," 2011.

[4] Def Stan 00-56, "Safety management requirements for defence systems," 2007.

[5] J. Westman, M. Nyberg, and M. Törngren, "Structuring safety requirements in iso 26262 using contract theory," in *Proceedings of the 32Nd International Conference on Computer Safety, Reliability, and Security - Volume 8153*, ser. SAFECOMP 2013. New York, NY, USA: Springer-Verlag New York, Inc., 2013, pp. 166–177. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40793-2_16

[6] J. Westman and M. Nyberg, "Environment-Centric Contracts for Design of Cyber-Physical Systems," in *Model-Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, J. Dingel, W. Schulte, I. Ramos, S. Abrahao, and E. Insfran, Eds. Springer International Publishing, 2014, vol. 8767, pp. 218–234.

[7] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple Viewpoint Contract-Based Specification and Design," in *Formal Methods for Components and Objects*, ser. Lecture Notes in Computer Science, F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Springer Berlin Heidelberg, 2008, vol. 5382, pp. 200–225. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92188-2_9

[8] A. L. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems," *European Journal of Control*, vol. 18, no. 3, pp. 217–238, 2012.

[9] SPEEDS, "SPEculative and Exploratory Design in Systems Engineering," 2006-2009. [Online]. Available: http://www.speeds.eu.com/

[10] A. Soderberg and R. Johansson, "Safety contract based design of software components," in *Software Reliability Engineering Workshops (ISSREW), 2013 IEEE Int. Symposium on*, Nov 2013, pp. 365–370.

[11] W. Damm, B. Josko, and T. Peinkamp, "Contract Based ISO CD 26262 Safety Analysis," in *Safety-Critical Systems, 2009*. SAE, 2009.

[12] B. Meyer, "Applying "Design by Contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992. [Online]. Available: http://dx.doi.org/10.1109/2.161279

[13] C. B. Jones, "Specification and Design of (Parallel) Programs," in *Information Processing 83*, ser. IFIP Congress Series, R. E. A. Mason, Ed., vol. 9, IFIP. Paris, France: North-Holland, Sep. 1983, pp. 321–332.

[14] A. Pnueli, "In transition from global to modular temporal reasoning about programs," in *Logics and Models of Concurrent Systems*, ser. NATO ASI Series, K. Apt, Ed. Springer Berlin Heidelberg, 1985, vol. 13, pp. 123–144. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-82453-1_5

[15] G. Shurek and O. Grumberg, "The modular framework of computer-aided verification," in *Computer-Aided Verification*, ser. Lecture Notes in Computer Science, E. Clarke and R. Kurshan, Eds. Springer Berlin Heidelberg, 1991, vol. 531, pp. 214–223. [Online]. Available: http://dx.doi.org/10.1007/BFb0023735

[16] M. Abadi and L. Lamport, "Composing specifications," *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 1, pp. 73–132, Jan. 1993. [Online]. Available: http://doi.acm.org/10.1145/151646.151649

[17] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran, "Mocha: Modularity in model checking," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Hu and M. Vardi, Eds. Springer Berlin Heidelberg, 1998, vol. 1427, pp. 521–525. [Online]. Available: http://dx.doi.org/10.1007/BFb0028774

[18] H. Giese, "Contract-based Component System Design," in *Thirty-Third Annual Hawaii Int. Conf. on System Sciences (HICSS-33), Maui*. IEEE Press, 2000.

[19] X. Sun, P. Nuzzo, C.-C. Wu, and A. Sangiovanni-Vincentelli, "Contract-based System-Level Composition of Analog Circuits," in *Design Automation Conf., 2009. DAC '09. 46th ACM/IEEE*, july 2009, pp. 605 –610.

[20] G. Goessler and J.-B. Raclet, "Modal contracts for component-based design," in *Proc. of the 2009 7th IEEE Int. Conf. on Software Eng. and Formal Methods*, ser. SEFM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 295–303. [Online]. Available: http://dx.doi.org/10.1109/SEFM.2009.26

[21] B. Delahaye, B. Caillaud, and A. Legay, "Probabilistic contracts: A compositional reasoning methodology for the design of systems with stochastic and/or non-deterministic aspects," *Form. Methods Syst. Des.*, vol. 38, no. 1, pp. 1–32, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1007/s10703-010-0107-8

[22] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. G. Larsen, "Contracts for System Design," INRIA, Rapport de recherche RR-8147, Nov. 2012. [Online]. Available: http://hal.inria.fr/hal-00757488

[23] S. Bauer, A. David, R. Hennicker, K. Guldstrand Larsen, A. Legay, U. Nyman, and A. Wąsowski, "Moving from specifications to contracts in component-based design," in *Fundamental Approaches to Software Eng.*, ser. Lec. Notes in Computer Science, J. Lara and A. Zisman, Eds. Springer Berlin Heidelberg, 2012, vol. 7212, pp. 43–58. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28872-2$_$3

[24] S. Quinton and S. Graf, "Contract-based verification of hierarchical systems of components," in *Software Engineering and Formal Methods, 2008. SEFM '08. Sixth IEEE International Conference on*, nov. 2008, pp. 377 –381.

[25] I. Bate, R. Hawkins, and J. McDermid, "A contract-based approach to designing safe systems," in *Proc. of the 8th Australian Workshop on*

Safety Critical Sys. and SW - Volume 33, ser. SCS '03. Australian Computer Society, Inc., 2003, pp. 25–36.

[26] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970. [Online]. Available: http://doi.acm.org/10.1145/362384.362685

[27] ISO/IEC/IEEE 42010, "System and software eng. - Architecture description," Geneva, Switzerland, 2011.

[28] J. Misra and K. Chandy, "Proofs of Networks of Processes," *Software Engineering, IEEE Transactions on*, vol. SE-7, no. 4, pp. 417–426, 1981.

[29] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[30] E. W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975. [Online]. Available: http://doi.acm.org/10.1145/360933.360975

[31] J. Westman and M. Nyberg, "A Reference Example on the Specification of Safety Requirements using ISO 26262," in *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*, M. ROY, Ed., France, Sep. 2013, p. NA. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00848610

[32] ——, "Extending Contract theory with Safety Integrity Levels," in *HASE, 2015 IEEE 16th Int. Symposium on*, Jan 2015, pp. 85–92.

Appendix I Industrial Case Study – The Fuel Level Display-system

This section shows an explicit use of the concepts presented in Section 2- 3 by structuring and specifying *safety requirements*, i.e. requirements that have been assigned a SIL [32], on an industrial system. The safety requirements will be structured as proposed in ISO 26262 - a functional safety standard for the automotive industry. The industrial system is a Fuel Level Display (FLD)-system, installed on all heavy trucks manufactured by Scania. In accordance with ISO 26262, a top-level safety requirement, i.e. *a safety goal*, for the FLD-system is broken down all the way into SW and HW Safety Requirements on the C-code implementation and associated HW.

Specifically, this is done by modeling the FLD-system and its environment as an architecture $\mathscr{A}_{FLDSys}$ in accordance with Section 2.2. Contracts are formed for elements in $\mathscr{A}_{FLDSys}$ in accordance with Section 3. The contracts, expressing relations between safety requirements on the elements in $\mathscr{A}_{FLDSys}$ in accordance with Section 3.1, are structured using a contract structure as presented in Section 3.3.

I .1 Architecture of the FLD-System

The FLD-system is a safety-critical system and this section will consider the actual C-code that is compiled and flashed onto the produced vehicles. The basic functionality of the FLD-system is to provide an estimate of the fuel volume in the fuel tank to the driver. The functionality is distributed across three ECU (Electric Control Unit)-systems, i.e. an ECU with sensors and actuators, in the E/E-system of the vehicle $\mathbb{E}_{Vehicle}$: Engine Management System (EMS) $\mathbb{E}_{EMS}$, Instrument Cluster (ICL) $\mathbb{E}_{ICL}$, and Coordinator (COO) $\mathbb{E}_{COO}$. The ECU-systems also interact with the fuel tank $\mathbb{E}_{FuelTank}$ and the parking brake system $\mathbb{E}_{pBrakeSys}$ that both are outside of the E/E-system of the truck. The environment of the vehicle $\mathbb{E}_{Vehicle}$ in $\mathscr{A}_{FLDSys}$ solely consists of a "driver" $\mathbb{E}_{Driver}$ with an interface identical to the vehicle $\mathbb{E}_{Vehicle}$.

There are several architectural variants of the FLD-system, e.g. variability in fuel tanks, types of sensors, etc. Due to space restrictions, only one type of architecture is considered here. The considered variant is shown in Figure 12 where all port variables and elements that are not relevant have been removed for reasons of readability. The sharing of port variables is ei-

ther visualized by connecting port variables with a line or by the appearance of port variables on several edges of rectangles.

COO $\mathbb{E}_{COO}$ estimates the fuel volume $actualFuelVolume[\%]$ in the tank $\mathbb{E}_{FuelTank}$ by a Kalman filter. The input signals to $\mathbb{E}_{COO}$ are: the position $sensedFuelLevel[\%]$ of a floater in the fuel tank $\mathbb{E}_{FuelTank}$, as sensed by the fuel sensor $\mathbb{E}_{fuelSensor}$; and the Controller Area Network (CAN) signal $FuelRate[l/h]$ in the message $FuelEconomy$, transmitted on $CAN1$ from EMS $\mathbb{E}_{EMS}$. The CAN signal $FuelRate[l/h]$ is an estimate of the current fuel consumption. The estimated fuel volume is transmitted on $CAN2$ as the CAN signal $FuelLevel[\%]$ in the CAN message $DashDisplay$. The CAN message is received by ICL $\mathbb{E}_{ICL}$ where a fuel gauge $indicatedFuelVolume$-$[\%]$ in the display presents the information to the driver.

A development according to ISO 26262 revolves around an *item*, i.e. *"a system that implements a function at a vehicle level"* [2]. For the analysis in this section, COO $\mathbb{E}_{COO}$ is chosen to be the item. In Figure 12, the architecture of the SW $\mathbb{E}_{SW}$ of the ECU $\mathbb{E}_{ECU}$ of $\mathbb{E}_{COO}$ is shown, as well as the ECU-HW $\mathbb{E}_{HW}$, the fuel sensor $\mathbb{E}_{fuelSensor}$, and the relevant elements in the environment of the item in $\mathscr{A}_{FLDSys}$.

The SW architecture of COO is structured as follows: the APPLication (APPL) layer consists of SW components that implement the high level functionality of the ECU $\mathbb{E}_{ECU}$; the MIDDleware (MIDD) layer contains the SW components in charge of controlling the I/O, e.g. sensors and actuators, connected to the ECU and the encoding/decoding of CAN messages; and the Basic Input/Output System (BIOS) layer contains the SW components that manages the low-level interaction with the executing platform, i.e. the ECU HW $\mathbb{E}_{HW}$.

For further understanding, functions inside of the different elements in the SW architecture are also shown in Figure 12, as well as data and control flow between the functions. Each function that is shown in Figure 12 is part of an element and is associated with a *function trigger*, i.e. a boolean port variable of the element where the port variable shares the same identifier as the function. Due to space restrictions, the identifiers of the function triggers are not shown in Figure 12, and each function trigger that is associated with a main function, identified as `funcID_20ms` in Figure 12, is not shown at all.

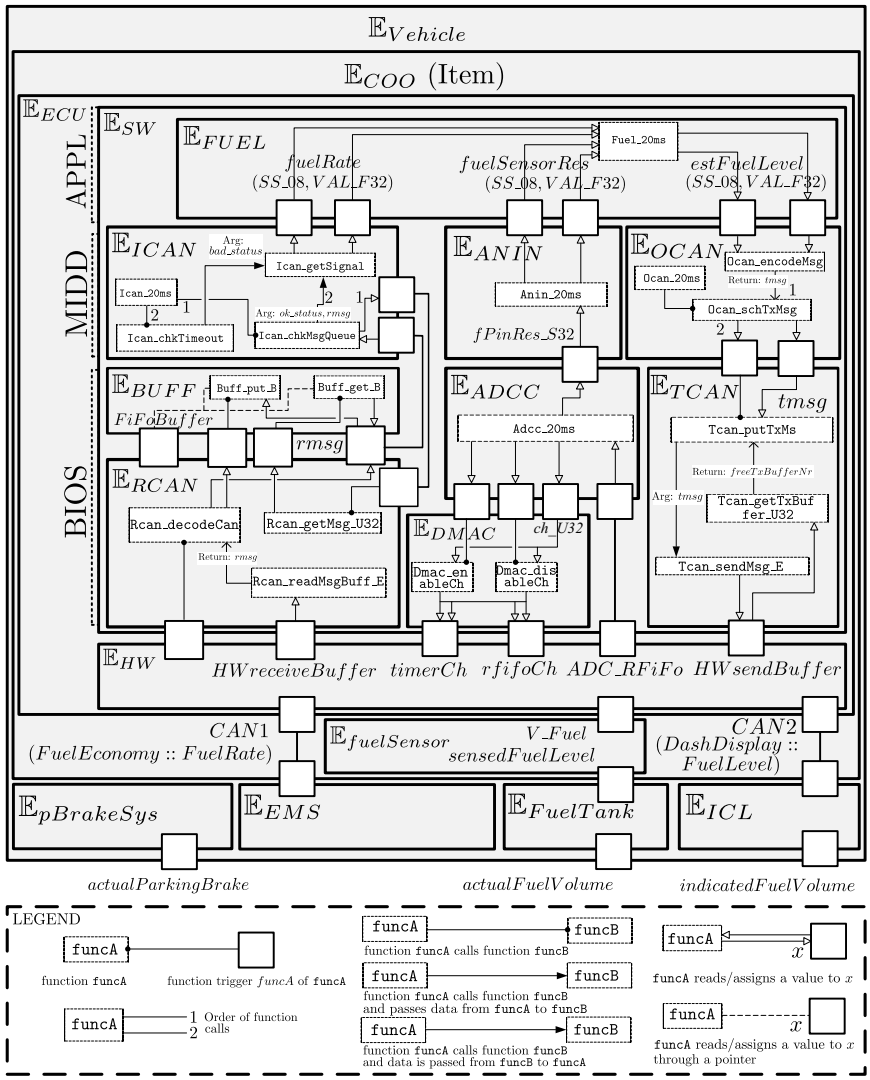


Figure 12. The architecture $\mathscr{A}_{FLDSys}$ of the FLD-system where only the relevant port variables and elements are shown. Different relations, e.g. data and control flow, between different SW functions and port variables are explained in the legend at the bottom of the figure. *NOTE:* The architecture and the naming are modified to protect the integrity of the implementation. The original complexity and intent of the architecture is, however, sustained.

I .2 Specification and Structuring of Safety Requirements

In this section, the safety goal and safety requirements are specified and structured in parallel to the architecture $\mathscr{A}_{FLDSys}$ of the FLD-system using a contract structure for $\mathscr{A}_{FLDSys}$ as shown in Figure 13. The specification of a guarantee of a contract for an element in $\mathscr{A}_{FLDSys}$ is made by considering the intended responsibility of the element in the context as a whole. This means that the guarantee, i.e. a safety requirement on the element, and also the assumptions, will not necessarily be limited to the interface of the element.

Regarding the representation of requirements, port variables are referred to using the format '$name[unit]$' or simply '$name$', such as e.g. $actualFuel\text{-}Volume[\%]$ and $FifoBuffer$. To pair a requirement with assumptions, a reference is made to one, or a set of requirement on elements in the environment, by writing '{SR#}' besides the requirement.

Different hierarchical levels of requirements as described in ISO 26262, i.e. safety goals, Functional Safety Requirements (FSRs), Technical Safety Requirements (TSRs), and Hardware and Software Safety Requirements (HWSRs/SSRs) are mapped to the type of port variable referred to in the requirements. That is, if a requirement refers to port variables that model properties at a vehicle level or those shared between ECU-systems, it is considered to be an FSR. If both port variables with HW and SW properties are referred to, it is considered to be a TSR and if only port variables with e.g. only SW properties are referenced, it is considered to be a SSR, and so on.

As a limitation due to space restrictions, only requirements that are applicable when the ignition is on, are considered. In SW, that corresponds to only including requirements applicable during run-time. It is assumed that every main function, identified as `funcID_20ms` in Figure 12, is run every 20 ms and that the data-flow is exclusive to what is shown in Figure 12, i.e. no other element can read or write to a variable other than what is shown. Furthermore, upon triggering a function by setting its function trigger from false to true, the function is assumed to terminate (which means that the function trigger is again set to false) within a negligible time. The term '*corresponds to*' is used when two variables/values are approximately equal, e.g. when they only differ in type or deviate due to small delays.

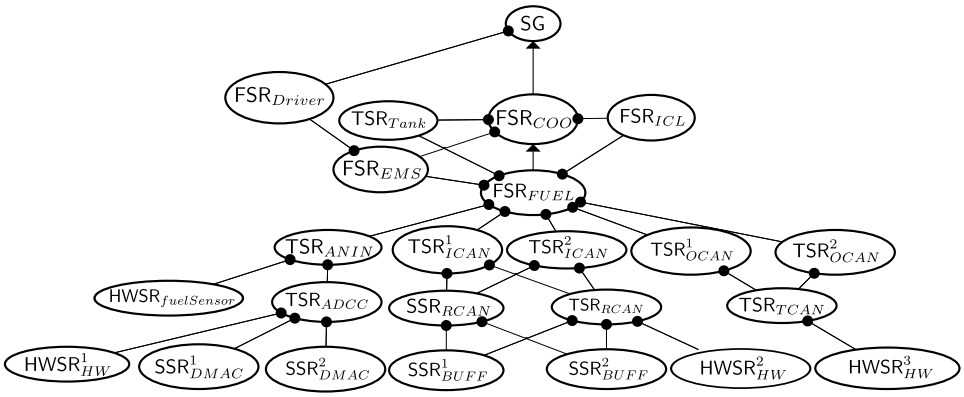


Figure 13. Contract structure for the architecture $\mathscr{A}_{FLDSys}$ where the structure of the elements are not shown for reasons of readability.

I .2.1 Specification of the Safety Goal

As shown in Table 1, the safety goal SG, under the responsibility of the vehicle $\mathbb{E}_{Vehicle}$, expresses that: *the indicated fuel volume, shown by the gauge, shall not be greater than the fuel volume in the tank while the parking brake is not applied.* The state when the vehicle $\mathbb{E}_{Vehicle}$ is parked is hence considered as a safe state of the vehicle. The safety goal SG is also shown as a node in the contract structure shown in Figure 13.

I .2.2 Safety Requirements on COO (the item) and its environment

As shown in Figure 13, the safety goal SG has an incoming "Assumptions of" arc from the FSR FSR_{Driver}, which means that FSR_{Driver} is an *assumption* of the contract $(\{\mathsf{FSR}_{Driver}\}, \mathsf{SG})$ for $\mathbb{E}_{Vehicle}$ in accordance with Section 3.3. This means that the safety goal is under the responsibility of the vehicle $\mathbb{E}_{Vehicle}$, but only if the driver $\mathbb{E}_{Driver}$ fulfills the assumption FSR_{Driver}, i.e. that *the driver does not refuel the vehicle while the parking brake is not applied*, as expressed in Table 2.

In accordance with Section 3.3, the arc from the FSR FSR_{COO} to the safety goal SG in Figure 13, means that the intent is that FSR_{COO} fulfills SG. As can be seen in Table 1, these two requirements are the same, which means that the responsibility of SG is fully delegated from the vehicle $\mathbb{E}_{Vehicle}$ to

Table 1. Safety requirements on COO $\mathbb{E}_{COO}$ (the item) and Safety Goal

SG	IF $actualParkingBrake[Bool]$ is false, THEN $indicatedFuelVolume$[%], shown by the fuel gauge, is less than or equal to $actualFuelVolume$[%]. $\{\mathsf{FSR}_{Driver}\}$
FSR_{COO}	IF $actualParkingBrake[Bool]$ is false, THEN $indicatedFuelVolume$[%], shown by the fuel gauge, is less than or equal to $actualFuelVolume$[%]. $\{\mathsf{TSR}_{Tank}, \mathsf{FSR}_{EMS}, \mathsf{FSR}_{ICL}\}$

Table 2. Safety requirements on the environment of the item (ICL $\mathbb{E}_{ICL}$, Tank $\mathbb{E}_{FuelTank}$, Driver $\mathbb{E}_{Driver}$, and EMS $\mathbb{E}_{EMS}$)

FSR_{Driver}	IF $actualParkingBrake[Bool]$ is false, THEN the derivative of $actualFuelVolume$[%] is less than or equal to 0.
TSR_{Tank}	The position of the floater $sensedFuelLevel$[%], sensed by the fuel sensor, does not deviate more than ±10% from $actualFuelVolume$[%] in the fuel tank.
FSR_{EMS}	IF $actualParkingBrake[Bool]$ is false AND $CAN1$ is equal to CAN message $FuelEconomy$ AND it has not passed more than $0.3s^1$ since the last time $CAN1$ was equal to $FuelEconomy$, THEN CAN signal $FuelRate[litres/h]$ in $FuelEconomy$ does not deviate more than ±1% from the derivative of $actualFuelVolume$[%]; OR $FuelRate[litres/h]$ is equal to $0xFE$ (error). $\{\mathsf{FSR}_{Driver}\}$
FSR_{ICL}	IF it has not passed more than $1s^1$ since the last time $CAN2$ was equal to CAN message $DashDisplay$ containing CAN signal $FuelLevel$[%] that is not equal to $0xFE$ (error), THEN $indicatedFuelVolume$[%], shown by the fuel gauge, corresponds to $FuelLevel$[%]. Otherwise, $indicatedFuelVolume$[%] is equal to 0.

COO $\mathbb{E}_{COO}$. However, the FSR FSR_{COO} is only under the responsibility of $\mathbb{E}_{COO}$, if the environment of $\mathbb{E}_{COO}$ fulfills the requirements TSR_{Tank}, FSR_{EMS}, and FSR_{ICL}, as shown in Figure 13.

As shown in Table 2, the requirements TSR_{Tank} and FSR_{EMS}, express the following in the nominal case, respectively: *the fuel sensor* $\mathbb{E}_{fuelSensor}$ *is correctly installed in the tank* $\mathbb{E}_{FuelTank}$; and *EMS* $\mathbb{E}_{EMS}$ *provides an accurate estimate of the fuel consumption, when driving.* The FSR FSR_{ICL}, under the responsibility of ICL $\mathbb{E}_{ICL}$, expresses in the nominal case that: *the indicated fuel volume, shown by the gauge, shall correspond to the estimated fuel volume, transmitted onto* $CAN2$ *by* $\mathbb{E}_{COO}$. As further shown in Figure 13, the assumption FSR_{Driver} of the contract for the vehicle, is also an assumption of the contract containing FSR_{EMS}, under the responsibility of $\mathbb{E}_{EMS}$.

Table 3. Safety requirements on APPL SW and MIDD SW components

FSR_{FUEL}	IF $actualParkingBrake[Bool]$ is false, THEN $indicatedFuelVolume[\%]$, shown by the fuel gauge, is less than or equal to $actualFuelVolume[\%]$. $\{\mathsf{TSR}_{Tank}, \mathsf{FSR}_{EMS}, \mathsf{FSR}_{ICL}, \mathsf{TSR}_{ANIN}, \mathsf{TSR}^{1-2}_{ICAN}, \mathsf{TSR}^{1-2}_{OCAN}\}$
TSR_{ANIN}	$fuelSensorRes\_Val\_F32[\%]$ corresponds to the floater position $sensedFuelLevel[\%]$, sensed by the fuel sensor; OR $fuelSensorRes\_SS\_U08[Enum]$ has the value ERR. $\{\mathsf{TSR}_{ADCC}, \mathsf{HWSR}_{fuelSensor}\}$
TSR^{1}_{ICAN}	IF it has not passed more than $0.3s$[1] since the last time $CAN1$ was equal to CAN message $FuelEconomy$ containing CAN signal $FuelRate[l/h]$ that is not equal to $0xFE$ (error), THEN $fuelRate\_Val\_F32[l/h]$ corresponds to $FuelRate[l/h]$. $\{\mathsf{SSR}_{RCAN}, \mathsf{FSR}_{RCAN}\}$
TSR^{2}_{ICAN}	IF CAN signal $FuelRate[l/h]$ in CAN message $FuelEconomy$ was equal to $0xFE$ (error) the last time $CAN1$ was equal to $FuelEconomy$ OR it has passed more than $0.3s$[1] since the last time $CAN1$ was equal to $FuelEconomy$, THEN $fuelRate\_SS\_U08[Enum]$ is equal to ERR. $\{\mathsf{SSR}_{RCAN}, \mathsf{TSR}_{RCAN}\}$
TSR^{1}_{OCAN}	IF $CAN2$ is equal to CAN message $DashDisplay$ containing CAN signal $FuelLevel[\%]$ that is equal to $0xFE$ (error) AND it has not passed more than $1s$[1] since the last time $CAN2$ was equal to $DashDisplay$, THEN $estFuelLevel\_SS\_U08[Enum]$ corresponds to ERR. $\{\mathsf{TSR}_{TCAN}\}$
TSR^{2}_{OCAN}	IF $CAN2$ is equal to CAN message $DashDisplay$ containing CAN signal $FuelLevel[\%]$ that is not equal to $0xFE$ (error) AND it has not passed more than $1s$[1] since the last time $CAN2$ was equal to $DashDisplay$, THEN $FuelLevel[\%]$ corresponds to $estFuelLevel\_Val\_F32[\%]$. $\{\mathsf{TSR}_{TCAN}\}$

[1]For confidentiality reasons, the values are either modified or not provided.

I .2.3 Safety Requirements on Application and Middleware SW

In Table 3, the safety requirements FSR_{FUEL}, TSR_{ANIN}, $\mathsf{TSR}^{1-2}_{ICAN}$, and $\mathsf{TSR}^{1-2}_{OCAN}$ on the SW components FUEL $\mathbb{E}_{FUEL}$, ANIN $\mathbb{E}_{ANIN}$, ICAN $\mathbb{E}_{ICAN}$ and OCAN $\mathbb{E}_{OCAN}$, respectively, are presented. The SW components in the MIDD SW provide the APPL SW with SW-signals that correspond to readings from sensors and CAN signals and also encode SW-signals from the APPL SW into CAN messages.

As can be seen in Table 1 and Table 3, the FSR FSR_{FUEL} on FUEL is the same as FSR_{COO}, which means that the responsibility of FSR_{COO} is delegated to FUEL, given that the environment of $\mathbb{E}_{FUEL}$ fulfills the requirements TSR_{Tank}, FSR_{EMS}, and FSR_{ICL}, and furthermore TSR_{ANIN}, $\mathsf{TSR}^{1-2}_{ICAN}$, and $\mathsf{TSR}^{1-2}_{OCAN}$ as shown in Figure 13.

The TSR TSR_{ANIN} expresses that the input signal $fuelSensorRes\_Val\_F32[\%]$ corresponds to the position of the floater $sensedFuelLevel[\%]$, or the status signal $fuelSensorRes\_SS\_U08[Enum]$ has the value ERR. The TSRs $\mathsf{TSR}^{1-2}_{ICAN}$ expresses that the input signal $fuelRate\_Val\_F32[litres$-

Table 4. Safety requirements on BIOS SW components

TSR_{ADCC}	$fPinRes\_s32[mV]$ corresponds to the voltage value $V\_Fuel[mV]$. $\{\mathsf{HWSR}^1_{HW}, \mathsf{SSR}^1_{DMAC}, \mathsf{SSR}^2_{DMAC}\}$
TSR_{RCAN}	IF $CAN1$ is equal to a CAN message $rmsg$, THEN $rmsg$ is in $FiFoBuffer$ within 20ms. $\{\mathsf{HWSR}^2_{HW}, \mathsf{SSR}^{1-2}_{BUFF}\}$
SSR_{RCAN}	IF the oldest message in $FiFoBuffer$ has PGN $0xFEF2$ when $Rcan\_getRxMsg\_U32$ transitions from false to true, THEN $rmsg$ corresponds to $FuelEconomy$ when $Rcan\_getRxMsg\_U32$ transitions from true to false. $\{\mathsf{SSR}^{1-2}_{BUFF}\}$
TSR_{TCAN}	IF $tmsg$ has PGN $0xFEFC$ when $Tcan\_putTxMsg\_E$ transitions from false to true, THEN $DashDisplay$ corresponds to $tmsg$ the next time $CAN2$ is equal to $DashDisplay$. $\{\mathsf{HWSR}^3_{HW}\}$
SSR^1_{DMAC}	The DMA channel that corresponds[2] to ch_U32 when $Dmac\_enableCh$ transitions from false to true, is enabled when $Dmac\_enableCh$ transitions from true to false.
SSR^2_{DMAC}	The DMA channel that corresponds[2] to ch_U32 when $Dmac\_disableCh$ transitions from false to true, is disabled when $Dmac\_disableCh$ transitions from true to false.
SSR^1_{BUFF}	The value of $rmsg$ when $Buff\_put\_B$ transitions from false to true, is added to $FiFoBuffer$ before $Buff\_put\_B$ transitions from true to false.
SSR^2_{BUFF}	When $Buff\_get\_B$ transitions from true to false, $rmsg$ is the oldest message in $FiFoBuffer$.

[2] $timerCh\_U32$ and $rfifoCh\_U32$ correspond to channels $timerCh$ and $rfifoCh$, respectively.

$/h]$ to FUEL corresponds to CAN signal $FuelRate[l/h]$ in CAN message $FuelEconomy$ in case $FuelRate[l/h]$ is not equal to $0xFE$ (error). In case $FuelRate[l/h]$ is equal to $0xFE$ (error) or if the signal was expected sooner, then $fuelRate\_SS\_U08[Enum]$ is equal to ERR. The TSRs $\mathsf{TSR}^{1-2}_{OCAN}$ expresses that the output signal $estFuelLevel\_Val\_F32[\%]$ to FUEL is transmitted on $CAN2$ as the signal $FuelLevel[\%]$ in CAN message $DashDisplay$ if $estFuelLevel\_SS\_U08[Enum]$ is not equal to ERR. In case $estFuelLevel\_SS\_U08[Enum]$ is equal to ERR, then $FuelLevel[\%]$ is equal to $0xFE$.

I .2.4 Safety Requirements on Basic Input/Output System SW and HW Components

In Table 4, safety requirements TSR_{TCAN}, SSR_{RCAN}, TSR_{ADCC}, SSR^{1-2}_{DMAC}, and SSR^{1-2}_{BUFF} on the SW components TCAN $\mathbb{E}_{TCAN}$, RCAN $\mathbb{E}_{RCAN}$, ADCC $\mathbb{E}_{ADCC}$, DMAC $\mathbb{E}_{DMAC}$ and BUFF $\mathbb{E}_{BUFF}$, respectively, are pre-

Table 5. Safety requirements on COO ECU HW or HW components

$\mathsf{HWSR}_{fuelSensor}$	The fuel sensor converts the floater position $sensedFuelLevel[\%]$ into a voltage value $V\_Fuel[mV]$ according to table Y[3]; OR $3000 < V\_Fuel[mV]$ OR $V\_Fuel[mV] < 200$[3].
HWSR^1_{HW}	IF the DMA channels $timerCh$ AND $rfifoCh$ are enabled for approx. $20ms$, THEN a RAW value of $V\_Fuel[mV]$ is available in ADC_RFIFO.
HWSR^2_{HW}	IF $CAN1$ is equal to a CAN message $rmsg$, THEN $rmsg$ is in $HWreceiveBuffer$ AND $Rcan\_decodeCan$ is set to true.
HWSR^3_{HW}	IF a CAN message $tmsg$ is the oldest message in $HWsendBuffer$, THEN $CAN2$ is equal to $tmsg$, the next time $CAN2$ is equal to a message in $HWsendBuffer$.

[3]For confidentiality reasons, the values are either modified or not provided.

sented. The SW components in the BIOS-layer provide the MIDD-layer with SW-signals that correspond to voltage values at the input pins for analogue sensors and manages the HW/SW interaction.

As shown in Figure 13, safety requirements TSR_{ADCC} and $\mathsf{HWSR}_{fuelSensor}$ are assumptions of the contract containing the guarantee TSR_{ANIN}, under the responsibility of $\mathbb{E}_{ANIN}$. The TSR TSR_{ADCC} expresses that the SW-signal $fPinRes\_s32[mV]$ corresponds to a voltage value $V\_Fuel[mV]$ at one of the input pins of the ECU $\mathbb{E}_{ECU}$ and the HWSR $\mathsf{HWSR}_{fuelSensor}$ expresses that the fuel sensor $\mathbb{E}_{fuelSensor}$ either provides the intended values or values that are out-of-range, as shown in Table 5.

However, the TSR TSR_{ADCC} is only under the responsibility of $\mathbb{E}_{ADCC}$, given that the environment of $\mathbb{E}_{ADCC}$ fulfills the assumptions HWSR^1_{HW} and $\mathsf{SSR}^{1-2}_{DMAC}$ as shown in Figure 13. The assumption HWSR^1_{HW} expresses a requirement on the ECU HW $\mathbb{E}_{HW}$, i.e. that a RAW Analog/Digital Converter (ADC) value is available if the ADC is allowed to sample for $20ms$, i.e. an execution tick, as shown in Table 5. In order for ADCC $\mathbb{E}_{ADCC}$ to control the ADC, it has to enable/disable Direct Memory Access (DMA)-channels by calling functions in DMAC $\mathbb{E}_{DMAC}$. Hence, as stated in the assumptions $\mathsf{SSR}^{1-2}_{DMAC}$, under the responsibility of DMAC $\mathbb{E}_{DMAC}$, the DMA-channels are enabled/disabled by calling the functions `Dmac_disableCh`(ch_U32) and `Dmac_enableCh`(ch_U32), respectively, where appropriate values of the argument ch_U32 match specific DMA channels.

In Figure 13, the TSR TSR_{RCAN} and the SSR SSR_{RCAN} are assumptions of both the contracts containing the guarantees TSR^1_{ICAN} and TSR^2_{ICAN}.

The SSR SSR_{RCAN} expresses that if the oldest message $rmsg$ in the queue $FiFoBuffer$ has Parameter Group Number (PGN) number $OxFEF2$ when the function `Rcan_getRxMsg_U32` is called, then $rmsg$ corresponds to $FuelEconomy$. The TSR TSR_{RCAN} expresses that if $FuelEconomy$ has arrived within $20ms$, i.e. since the latest execution tick, then it has been placed in the queue $FiFoBuffer$.

As shown in Figure 13, the SSRs $\mathsf{SSR}^{1-2}_{BUFF}$ and HWSR^{2}_{HW} are the assumptions of the contract containing the guarantee TSR_{RCAN} and $\mathsf{SSR}^{1-2}_{BUFF}$ are the assumptions of the contract containing the guarantee SSR_{RCAN}. The SSRs $\mathsf{SSR}^{1-2}_{BUFF}$ express that the responsibility of $\mathbb{E}_{BUFF}$ is to manage the queue $FiFoBuffer$ and HWSR^{2}_{HW} expresses the requirement on the ECU HW $\mathbb{E}_{HW}$ that RCAN $\mathbb{E}_{RCAN}$ is notified whenever a new CAN message has arrived in $HWreceiveBuffer$.

The TSRs $\mathsf{TSR}^{1-2}_{OCAN}$ are under the responsibility of $\mathbb{E}_{OCAN}$ as shown in Figure 13, given that the environment of $\mathbb{E}_{OCAN}$ fulfills the TSR TSR_{TCAN}, under the responsibility of $\mathbb{E}_{TCAN}$. The TSR TSR_{TCAN} expresses that if the function `Tcan_putTxMsg_E` is called with an argument $tmsg$ with a PGN $0xFEFC$, then $DashDisplay$, corresponding to $tmsg$, is to be transmitted on $CAN2$, given that the ECU HW $\mathbb{E}_{HW}$ fulfills the HWSR $HWSR^{3}_{HW}$. The HWSR $HWSR^{3}_{HW}$ expresses that the messages placed in $HWsendBuffer$ are to be transmitted on $CAN2$ in the order in which they are placed in $HWsendBuffer$.

I .3 Discussion

As can be observed in Section I .2, each contract in Tables 1-5 allows a clear separation of responsibilities between an element in $\mathscr{A}_{FLDSys}$ and its environment by explicitly declaring what the element requires from its environment as assumptions, in order to ensure that the requirement is met. That is, if e.g. the fuel sensor $\mathbb{E}_{fuelSensor}$ is installed incorrectly, i.e. if the environment of $\mathbb{E}_{COO}$ does not fulfill the assumption TSR_{Tank} of the contract for $\mathbb{E}_{COO}$, then it cannot be ensured that $\mathbb{E}_{COO}$ meets its responsibility expressed by the FSR FSR_{COO}.

Regarding the specification of the requirements, references to port variables were made in compliance with the scoping conditions (a) and (b) of Theorem 2 (in Paper A). Considering the contract $(\{\mathsf{FSR}_{Driver}\}, \mathsf{SG})$ for

$\mathbb{E}_{Vehicle}$ as an example, since the set of variables

$$\{actualParkingBrake[Bool], actualFuelVolume[\%]\}$$

of the assumption FSR_{Driver} is a subset of the interface of $\mathbb{E}_{Driver}$, i.e. the environment of the vehicle $\mathbb{E}_{Vehicle}$ as mentioned in Section I .1, the condition (a) of Theorem 2 holds. Furthermore, since the set of variables constrained by SG is a subset of the interface of $\mathbb{E}_{Vehicle}$, the condition (b) of Theorem 2 holds.

The contract structure for $\mathscr{A}_{FLDSys}$ shown in Figure 13 organizes the safety requirements in Tables 1- 5 on elements in $\mathscr{A}_{FLDSys}$. The contract structure gives an overview of all the relations between the safety requirements and the possibility of tracing atomic safety requirements to the safety goal. The intent of the relations are well-defined in Theorem 1, which means that the exact conditions that need to be verified are given, in order to ensure that the vehicle meets its responsibility expressed by the safety goal SG.

Furthermore, since an assumption of a contract in the contract structure in Figure 13 corresponds to a guarantee of another contract in accordance with Section 3.1, the contract structure offers a compact representation of the dependencies between the requirements on the elements in $\mathscr{A}_{FLDSys}$. That is, if intermediate assumption nodes were instead used, then the contract structure in Figure 13 would have almost twice as many nodes. Moreover, the use of intermediate assumption nodes introduces an extra verification step than those presented in Theorem 1 in order to ensure that the property (1) holds.

Paper C

Formal Architecture Modeling of Sequential Non-Recursive C Programs

Jonas Westman • Mattias Nyberg • Joakim Gustavsson • Dilian Gurov

Submitted to *Science of Computer Programming*.

Abstract

To manage the complexity of C programs, *architecture models* are used as high-level descriptions, allowing developers to understand, assess, and manage the C programs without having to understand the intricate complexity of the code implementations. However, for the architecture models to serve their purpose, they must be *accurate* representations of the code implementations. In order to achieve this in practice, support is needed in the form of a stringent *mapping* from the C language to an architecture modeling formalism. Considering that there exists no such uniform mapping from the C language to Modeling Languages (MLs) such as SysML or UML and Architecture Description Languages (ADLs) such as AADL, modeling C programs using such languages is essentially ad-hoc. Therefore, a *unique* mapping is established from the domain of sequential non-recursive C programs to a domain of *formal* architecture models.

1 Introduction

Complexity of software (SW) programs in industry is high, for example, as stated in [1], the amount of SW in a premium car can be up to ten million lines of code – typically, this code is C code [2]. For developers to get an overall understanding of such C programs without having to comprehend the intricate complexity of the code implementations, *architecture models* are used as high-level *descriptions* [3] of the C programs; the present paper considers architecture models of *sequential non-recursive* C Programs. However, as stated in [4], *"to manage the complexity of developing, maintaining, and evolving a critical software-intensive system, its architecture description must be accurately and traceably linked to its implementation"*. In practice, in order to create architecture models that serve as *accurate* descriptions of C programs, support is needed in the form of a stringent *mapping* from the C language to an architecture modeling formalism.

Consider current approaches for architecture modeling, e.g. general purpose Modeling Languages (MLs) such as SysML [5] or UML [6] and Architecture Description Languages (ADLs) such as AADL [7]. Since there exists no uniform mapping from the C language to any of these ML/ADLs, modeling C programs using such languages is essentially ad-hoc. Models of C programs are used in approaches for formal verification of C programs (see e.g. [8, 9] or [10] for an overview) where a C program is translated into

a formal model that is fed into a tool for semi or fully automated analysis. However, such a formal model does not serve as an architecture model of a C program, but rather as a formalization of its execution semantics.

Thus, despite the fact that the C language is one of the most popular programming languages [11], there exists limited support for architecture modeling of C programs. This is contrasted by the fact that SW architecture models are in general considered to be essential for managing SW complexity [4]. This fact is also recognized in the automotive functional safety standard ISO 26262 [12] where the use of a SW architecture model is a hard requirement. However, not only is this the case, but the standard also enforces that SW safety specifications must be organized in accordance with the different levels of the architecture model, i.e. it is stated that *"steps of the integration and testing of the software elements correspond directly to the hierarchical architecture of the software"* [12]. Moreover, according to ISO 26262, global SW safety specifications must be decomposed all the way down to local safety specifications for SW units, and organized so that it can be inferred that the global SW specifications are satisfied, from verifying that the SW units satisfy their specifications; such an indirect verification approach is called *compositional verification* [13, 14].

In general, compositional verification allows verification of complex systems when it is infeasible, due to required effort/cost, to use a direct verification approach to ensure that the complete system satisfies a global specification. However, while providing a means to verify systems in general – including C programs – compositional verification requires the effort of iteratively decomposing specifications into lower-level specifications that can be satisfied by parts of a system. As made clear in [15], in order to support such an effort in practice, the specifications must be organized in parallel to an *architecture model* of a system where the model formally captures a structure of the system as a hierarchy of components with well-defined *interfaces* over which specifications are expressed.

Hence, to manage the complexity of C programs in general, and to support expressing and organizing specifications for C programs in particular, an architecture model of C programs is needed. Considering this need, as the main contribution, the present paper introduces a formal architecture model of sequential non-recursive C programs.

More specifically, a *unique* mapping is established from the domain of sequential non-recursive C programs to a domain of formal architecture models. The C programs are in the form of .c-files, hierarchically structured

into packages and layers that encapsulate e.g. Operating System (OS) services (e.g. scheduling) and communication services (Controller Area Network (CAN), I/O, etc.). Each level of the hierarchical structure of a C program, and also each C function, is mapped to an architectural component with a well-defined *interface* modeling C function and C variable identifiers in the C program code. Hence, the components and their interfaces provide a foundation both for organizing specifications in accordance with the structure of the C program and for expressing specifications in terms of identifiers in the C program.

As an additional contribution, considering that *manually* creating architecture models is an error-prone task in general [16], a study on the feasibility to automatically extract the proposed architecture model from a C program, is presented. This feasibility study revolves around the development of a prototype tool for architecture recovery [17], where the tool relies on the *LLVM Compiler Infrastructure* [18]. Not only does the study indicate the potential of automatically extracting the proposed architecture model, but it also highlights specific C program constructs that are, in practice, difficult to extract from the C program code. Hence, the study also serves as a guideline for what C program constructs to avoid, in order to be able to generate architecture models in general, and architecture models of the proposed type, in particular.

An architecture model of the proposed type can be compared with formal models in SW compositional frameworks (see e.g. [19–21] or [13, 14] for a surveys). Out of these, the works [19, 21] are the most similar to the present paper since the interfaces of components are explicit in these frameworks. However, the model in [19] does not support modeling encapsulation of local C variables and is tailored for capturing safety control flow properties, whereas the architecture model in the present paper is applicable for capturing functional properties in general. In contrast to [21], the architecture model in the present paper supports modeling global variables that are written to by different parts of a C program. Moreover, the works [19, 21] do not provide specific guidance for how a C program can be modeled whereas a C program is uniquely mapped to an architecture model in the present paper.

The paper is organized as follows. Section 2 presents general concepts relating to modeling and specification. Section 3 introduces concepts describing C programs and their structure. Section 4 presents the proposed mapping from C programs to architecture models. Section 5 presents the

feasibility study revolving around the development of the prototype tool for architecture recovery. Section 6 summarizes the paper and draws conclusions.

2 Preliminaries: Assertions, Elements, Architecture, and Contracts

This section summarizes relevant concepts originating from a compositional contract-based theory [22–25] for modeling and specifying *heterogeneous systems* [26–28], i.e. systems composed of parts from multiple domains, e.g. SW, HW, mechanical, electrical, etc. The theory [22–25] relies on a general formalism where interfaces are modeled as sets of variables and where *assertions*, i.e. sets of value sequences (runs), are used for expressing specifications and behaviors over the interfaces. The use of the theory [22–25] has been advocated in several contexts, e.g. Platform Based Design in [23], Model Based Design in [29], safety analyzes in [30], virtual integration and testing in [31], and for structuring safety requirements in [32–34].

The concepts presented in this section will serve as a theoretical basis for the proposed mapping from C programs to architecture models, as will be presented in Section 4. Notably, by relying on the general formalism [22–25] instead of considering a specific ML or ADL, the proposed architecture models and their specifications allow to be combined with models of, and specifications for, parts in other domains.

2.1 Assertions and Runs

In the following, assume a universal set of variables Ξ, a domain for each variable $x \in \Xi$, and a total order on Ξ.

2.1.1 Runs

Consider a subset $X \subseteq \Xi$ and an index set $I = \{0, 1, 2, \ldots, N \geq 0\}$, possibly infinite. A *run for* X *over* I, written $\omega_{X,I}$ or simply ω, is a sequence of values of the variables in X over the index set I. That is, a run is a sequence $(X_i)_{i \in I}$ where the i:th term X_i denotes a tuple $(x(i))_{x \in X}$ that is ordered according to the total order on Ξ and where $x(i)$ denotes the value of the variable x at index i. For example, a run can be a *trace* [35–38] or an *execution* as presented in [39].

Given a run $\omega_{X,I}$ and a set of variables $X' \subseteq X$, the *projection* [24, 25] of $\omega_{X,I}$ onto X', written $proj_{X'}(\omega_{X,I})$, is the run $(X'_i)_{i\in I}$ where each tuple X'_i is obtained by removing the value $x(i)$ of each variable $x \notin X'$ from the i:th tuple X_i in the run $\omega_{X,I}$. Note that projection is defined on a set of runs rather than a single run in [24, 25]. Furthermore, in essence, projection corresponds to the operator $\prod$ in relational algebra [40].

2.1.2 Assertions

Let Ω denote the set of all possible runs for Ξ over each index set in $\{\{0, 1, 2, \ldots, N\} \mid N \geq 0\}$. An *assertion* W is a, possibly empty, subset of Ω, i.e. $\mathsf{W} \subseteq \Omega$.

An assertion can be syntactically represented by constraints, e.g. by equations, inequalities, or logical formulas. For example, an assertion W' represented by the equation $u = v$, is the set of all runs in Ω where $u(i) = v(i)$ holds for each index i.

As a second example of how an assertion can be syntactically represented, consider that W'' is an assertion represented by the logic formula $a(i) = 0 \vee b(i) = 0$ where $i \in \{0, 1, \ldots, 10\}$ and where both variables a and b take values from $\{0, 1\}$. This means that the assertion W'' is the set of all possible runs for Ξ over $\{0, 1, \ldots, 10\}$ where, for each index i, at least one of a and b has the value 0.

The *extended projection* [24, 25] of W onto X is denoted $\widehat{proj}_X(\mathsf{W})$ and is the inverse projection of $\{proj_X(\omega) \mid \omega \in \mathsf{W}\}$ onto Ξ, i.e.

$$\widehat{proj}_X(\mathsf{W}) = \{\omega' \in \Omega \mid proj_X(\omega') \in \{proj_X(\omega) \mid \omega \in \mathsf{W}\}\} \ . \tag{1}$$

Conceptually, extended projection corresponds to the notion of *port elimination* [22] or *variable hiding* [41, 42] through existential quantification.

As an example of extended projection, consider an assertion represented by the equation $y = x$, where the assertion will be denoted as $\mathsf{W}_{y=x}$ for convenience. The assertion $\mathsf{W}_{y=x}$ is shown in Figure 1a where the x'-axis can be the axis of any variable in $\Xi \backslash \{x, y\}$ and where only one index is shown considering that $y = x$ is independent of the index. The set of projections of the runs in $\mathsf{W}_{y=x}$ onto $\{x\}$, i.e. the set of runs $\{proj_{\{x\}}(\omega) \mid \omega \in \mathsf{W}_{y=x}\}$, is shown in Figure 1b. The extended projection of $\mathsf{W}_{y=x}$ onto $\{x\}$, i.e. the assertion $\widehat{proj}_X(\mathsf{W}_{y=x})$, is the inverse projection of $\{proj_{\{x\}}(\omega) \mid \omega \in \mathsf{W}_{y=x}\}$. This assertion is shown in Figure 1c and is in fact the assertion Ω.

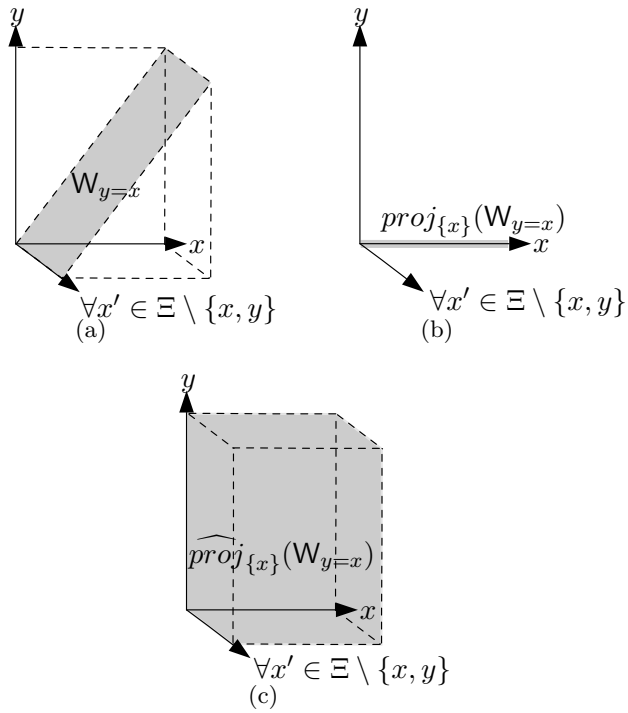


Figure 1. In a), b), and c), an assertion $\mathsf{W}_{y=x}$, the set of runs $\{proj_{\{x\}}(\omega) \mid \omega \in \mathsf{W}_{y=x}\}$, and the extended projection of $\mathsf{W}_{y=x}$ onto $\{x\}$ are shown, respectively.

As previously presented, assertions are sets of runs for the universal set of variables Ξ. However, as can be seen in previous examples, assertions can be represented by constraints specified over a subset of Ξ. For example, the equation $x = y$ representing the assertion $\mathsf{W}_{x=y}$ shown in Figure 1a, is specified simply over the set of variables $\{x, y\}$. For a given assertion, the following introduces a concept that distinguishes such a set of variables from the set of variables Ξ.

A variable x is constrained by an assertion W if it holds that

$$\widehat{proj}_{\Xi\setminus\{x\}}(\mathsf{W}) \neq \mathsf{W} \ . \tag{2}$$

For example, the set of variables constrained by $\mathsf{W}_{x=y}$, is the set $\{x, y\}$.

2.2 States and Transitions of Assertions

This section introduces terminology for describing assertions by *states* and *transitions*, rather than as a set of runs. This terminology will be frequently used throughout the paper.

A *state of an assertion* W is a run $\omega_{\Xi,I}$ that is a prefix of a run in W. That is, there exists a run $\omega \in \mathsf{W}$ such that for each $i \in I$, it holds that the i:th tuple Ξ_i of $\omega_{\Xi,I}$, is also the i:th tuple of ω. For the case where $\omega_{\Xi,I}$ is finite, there exists an index $i \in I$ such that $i \in I$ and $i+1 \notin I$. For such a case, let $x(\omega_{\Xi,I})$ denote the value $x(i) \in \Xi_i$ for a given variable $x \in \Xi$.

A pair of states $(\omega_{\Xi,I}, \omega_{\Xi,I+1})$ of an assertion W is a *transition of* W if $\omega_{\Xi,I}$ is a state of $\{\omega_{\Xi,I+1}\}$. A sequence of runs $(\omega^1, \omega^2, \ldots, \omega^M)$ is *a sequence of an assertion* if the pair (ω^j, ω^{j+1}) is a transition of the assertion, for each $j = 1, \ldots, M-1$.

A state of W is also an *end state of* W if the state is a run of W. A *terminating state of* W is an end state with no successors.

2.3 Elements and Architectures

This section first introduces the concept of an *element* [24, 25], which is an abstract concept that can be refined to model any part in general, such as a software, hardware, or physical part, as well as logical and functional design parts, e.g. a SysML [5] block or Simulink [43] subsystem.

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.

Note that in [22, 23], elements correspond to a Heterogeneous Rich Components (HRCs) [44].

A set of elements can be organized into an *architecture* [24, 25], which describes an hierarchical nesting of elements. This hierarchical nesting can be viewed as a rooted tree; thus, in the following, terminology from graph theory [45] will be borrowed to describe positions of elements in an architecture, relative to each other. The underlying principle is to combine individual behaviors using *intersection* [22, 23, 46] where the *sharing of port*

variables [22, 23, 47] between elements captures the interaction points between the elements.

Definition 1 (Architecture)**.** An *architecture* 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 $\mathsf{B} = \widehat{proj}_X(\bigcap_{i=1}^{N} \mathsf{B}_i)$ and $X \subseteq \bigcup_{i=1}^{N} X_i$; and

(b) if there is a child $\mathbb{E}' = (X', \mathsf{B}')$ and a non-descendent $\mathbb{E}'' = (X'', \mathsf{B}'')$ 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* is the set of elements $\{\mathbb{E}_j\}_{i=1}^{M}$ such that each $\mathbb{E}_j$ is either a sibling or a sibling of a proper ancestor of $\mathbb{E}$.

As expressed in part (a) of the definition, the individual behaviors of the children of an element $\mathbb{E}$ are combined and abstracted with intersection and extended projection onto the interface of $\mathbb{E}$. Part (b) of the definition expresses that if a port variable x is part of the interface of both a child of an element $\mathbb{E}$ and an element in the environment of $\mathbb{E}$, then x must also be part of the interface of $\mathbb{E}$.

To get a grasp of what an architecture is, as well as to give a preview of the proposed mapping from C programs to architecture models as will be presented in Section 4, Figure 2a shows an architecture model of a C program `count10` where the architecture contains elements with typed interfaces, called *components* that will be formally introduced in Section 4.1.

As shown in Figure 2b, the code of the C program `count10` is contained in the code of two files mod.c and mod2.c in Figure 3. In turn, the code in these two .c files consists of the definitions of C function identifiers `add`, `step`, and `main`, and the remaining code `c2` and `c8` in the code lines 2 and 8. Note that the presented code in Figure 3 has the sole purpose of illustrating concepts related to the mapping properties that will be presented in Section 4 – it does not serve as an example of how to program in C.

In the definition of `add`, the sum of the values of two of its formal parameters a and b is stored in a location referenced by the address is given by the value of a third formal parameter r of `add`. In the definition of `step`, the value of an integer 'counter', which is referenced by the address $0x40000000$ and is assumed to be writable, is increased by one step. This is achieved

by: first, assigning the value of the counter to an integer t by dereferencing an integer pointer c that is initialized to the address of the counter; second, increasing the value of t with one step by invoking `add` with the arguments `t`, `1`, and `@t`; and third, assigning the counter with the value of t by dereferencing c. A function call to `step` is invoked repeatedly in the definition of `main`, which, assuming that the counter is initialized to the value 0, triggers a step-wise count from 0 to 10. When the counter reaches 10, `main` returns.

In Figure 2a, the rectangles filled with gray and the boxes on their edges represent the elements in the architecture and their port variables, respectively, and where *a shared port variable* is represented either by connecting boxes with a line or by the same box being present on several edges of rectangles. The fact that a rectangle representing an element $\mathbb{E}'$ is within another rectangle representing an element $\mathbb{E}$, represents that $\mathbb{E}'$ is a proper descendant of $\mathbb{E}$.

The C program `count10`, the code in the files mod.c and mod2.c, the definitions of `step`, `add`, and `main`, and the remaining code `c2` and `c8`, are mapped to the components $\mathbb{E}_{count10}$, $\mathbb{E}_{mod}$, $\mathbb{E}_{mod2}$, $\mathbb{E}_{step}$, $\mathbb{E}_{add}$, $\mathbb{E}_{main}$, $\mathbb{E}_{c2}$, and $\mathbb{E}_{c8}$, respectively. The typing (a partitioning and labeling) of the port variables in the interfaces of the components, specifies a mapping from C function identifier declarations and C variable identifiers to port variables in the interfaces. The typing is represented in Fig. 2a by enclosing boxes representing port variables in dashed brackets and attaching symbols to boxes. The specifics regarding what the brackets and the symbols in Fig. 2a represent, will be presented in Section 4 where the proposed mapping from C programs to architecture models is also presented.

2.4 Organizing and Specifying Contracts for Compositional Verification

This section shows how contracts can be specified over element interfaces and decomposed in accordance with the organization of the elements into an architecture. Thus, the architecture provides support for decomposing specifications in order to verify a global specification using compositional verification. The support is given for an architecture in general, including the case of an architecture model of a C program, as will be presented in Section 4.

A specification for an element $\mathbb{E} = (X, \mathsf{B})$ is expressed as a *contract*, which is a tuple $(\mathcal{A}, \mathsf{G}, X)$ where $\mathcal{A} = \{\mathsf{A}_1, \ldots, \mathsf{A}_N\}$ and where each A_i

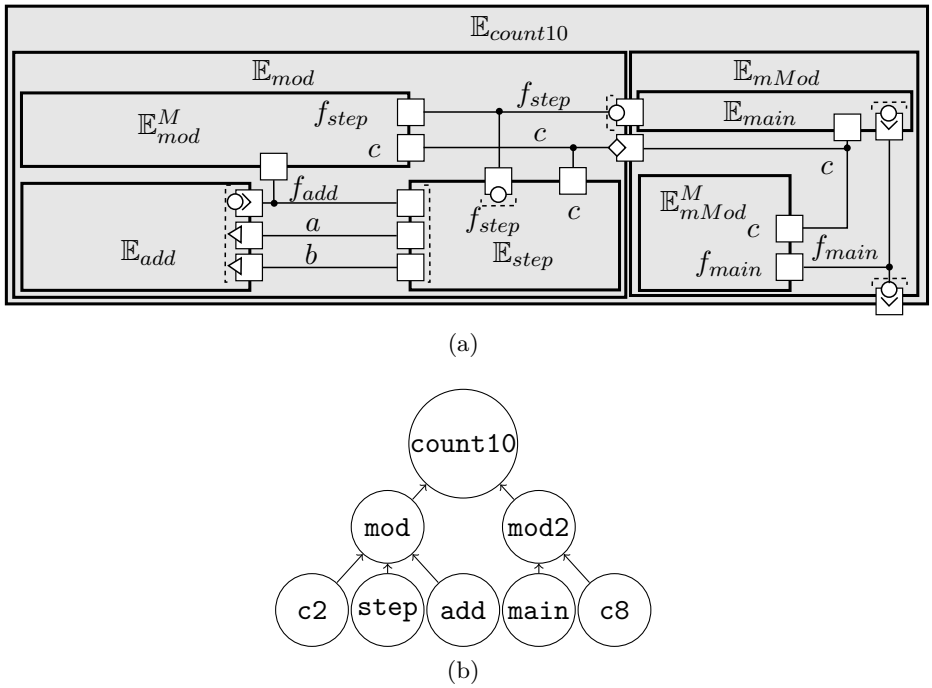


Figure 2. In a), an architecture model is shown of the structure of a C program `count10`, as shown in b).

and G are assertions called an *assumption* and the *guarantee*, respectively. The guarantee expresses an intended property under the responsibility of the element, given any architecture where the environment of the element enforces the assumptions. The element $\mathbb{E} = (X, \mathsf{B})$ *satisfies* the contract $(\mathcal{A}, \mathsf{G}, X)$ if

$$\mathsf{A}_1 \cap \ldots \cap \mathsf{A}_N \cap \mathsf{B} \subseteq \mathsf{G} \ . \tag{3}$$

Assume that a contract $(\{\mathsf{A}_1, \ldots, \mathsf{A}_N\}, \mathsf{G}, X)$ has been expressed for each element $\mathbb{E} = (X, \mathsf{B})$ in an architecture. For each element $\mathbb{E}$ that is not the root element, the intent is that there are assumptions $A_{i,1}, \ldots, A_{i,N_i}$ of the contract for the parent of $\mathbb{E}$ and guarantees $G_{i,1}, \ldots, G_{i,M_i}$ of contracts for siblings of $\mathbb{E}$ such that $N_i + M_i \geq 1$ and for each i:

$$\mathsf{A}_{i,1} \cap \ldots \cap \mathsf{A}_{i,N_i} \cap \mathsf{G}_{i,1} \cap \ldots \cap \mathsf{G}_{i,M_i} \subseteq \mathsf{A}_i \ . \tag{4}$$

Furthermore, for each non-leaf element $\mathbb{E}$, the intent is that there are guarantees $G_1, \ldots, G_M$ of contracts for elements that are children of $\mathbb{E}$, such that

```
1  // --------------mod.c-------------
2  int * c = (int *) 0x40000000; // Initialization
       of global pointer to counter at address 0
       x40000000
3  void add(int a, int b, int * r) // Writes sum of
       a+b to *r
4          {int res=a+b; *r = res;}
5  void step(void) // Increases counter by 1
6          {int t=*c; add(t,1,&t); *c=t;}
```

```
7  // --------------mod2.c------------
8  extern int * c; // Declaration of c
9  int main (void) // Step-wise counting from 0 up
       to 10
10         {while(*c<10){step();} return 0;}
```

Figure 3. Code of a C program `count10`, consisting of the code in two files mod.c and mod2.c.

$M \geq 1$ and

$$\mathsf{G}_1 \cap \ldots \cap \mathsf{G}_M \subseteq \mathsf{G} \,. \tag{5}$$

Consider forming a graph where the nodes represent the assumptions and the guarantees of the contracts, and the edges represent intended relations of type (4)-(5). In accordance with [24, 25] and ignoring a few technicalities that are presented in depth in [24], if such a graph is a Directed Acyclic Graph (DAG) where all the intended relations represented by the edges hold, then it follows that the root element satisfies its contract if the leaf elements satisfy their contracts. This allows to verify that the root element satisfies its contract using compositional verification.

3 C Programs

In accordance with the general definition of a program in [46], a *C program* is a specification, syntactically represented as C code and semantically represented as an assertion, expressing entire executions of the code. To make a clear distinction between the two representations, the terms *C program*

and *C program assertion* will be used to refer to the syntactic and semantic representations, respectively.

This section now proceeds by first describing the concept of C program assertions in more detail, followed by the introduction of other concepts describing a C program and the manner in which it is structured. The concepts will be frequently used in Section 4 that presents the proposed mapping from C programs to architecture models.

3.1 Expressions and C Program Assertion

An assertion W_{prog}, representing a C program, will be considered to be such that each initial state maps to an invocation of `main`, or more specifically, to a point right before code within the definition of `main` is executed. Referring to the grammar in [2], each non-initial state of W_{prog} maps to an evaluation of an *expression*, e.g. a function call, an assignment, or an identifier, represented by a node in an Abstract Syntax Tree (AST) of the C program. For example, Figure 4 shows an AST of the code lines 9-10 in Figure 3, where nodes that represent expressions are shown as rectangles. To be more exact, each non-initial state of W_{prog} specifies a point in an execution *right after* the mapped expression has been evaluated and where there exists a run of W_{prog} for each valid order of evaluating expressions.

In the following, it will be assumed that the C program eventually terminates and that the last code executed is of `main`. This means that each run in W_{prog} will be considered to be finite and that the end states of W_{prog}, which are also terminating states, map to an expression in the code of `main` and specify points where execution is terminated.

Furthermore, the C program assertion W_{prog} will be considered to constrain a single, but multi-dimensional, variable x_{prog}, called *the variable of the C program*, which holds information about storage locations. That is, in accordance with the relation (2), it holds that $\mathsf{W}_{prog} = \widehat{proj}_{\{x_{prog}\}}(\mathsf{W}_{prog})$. In the following, when referring to values at certain points in an execution, i.e. values of C variables or resulting from evaluating expressions, it is assumed that these values are in the C program variable x_{prog}.

3.2 C Variables, C Functions, and Identifiers

A C program is contained in a set of *translation units* [2], i.e. preprocessed .c files where all preprocessing directives, e.g. `#include`, as well as constants

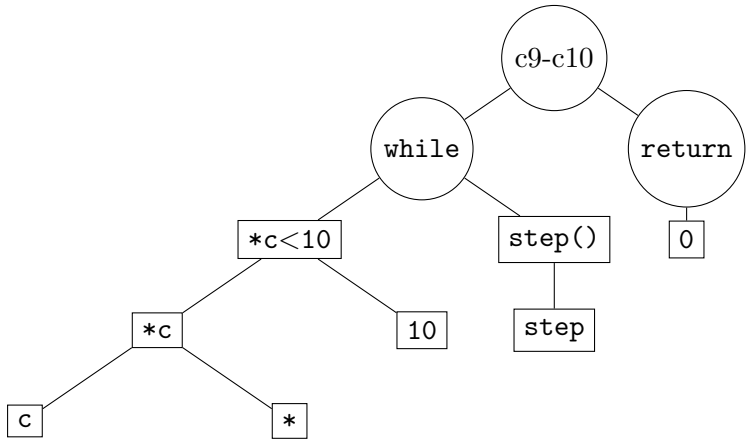


Figure 4. Abstract Syntax Tree of the code lines 9-10 in Figure 3.

and macro definitions, have been replaced by a preprocessor. For example, considering the code shown in Figure 3, except for the comments that will be removed by a preprocessor, the code in lines 2-6 is translation unit `mod` and the code in lines 8-10 is translation unit `mod2`.

In accordance with [2], a *C variable* is a storage location that has a value, called an *rvalue*, and is referenced by an *lvalue* (an address).

A *C variable identifier* is *defined* in a C program using a syntax such as `storage-class type name;` or a variant of it, see [2] for all variants. In a C program subset, called the *scope* of the identifier, the identifier evaluates to either the lvalue or rvalue of a C variable; the specified type, e.g. `int`, determines the rvalue.

In two different states of the C program assertion where these two states map to an evaluation of the identifier (as an expression) in its scope, it holds that the identifier may evaluate to the lvalue or rvalue of two *different* C variables. The storage class, e.g. `static`, as well as the C program context, determine a set of maximum length sequences of the C program assertion where the identifier may only evaluate to the lvalue or rvalue of the *same* C variable. Each such sequence is called a *life-time of a C variable with respect to the identifier*. In the following, a C variable will be said to be *associated with an identifier at a state* if the state is in the life-time of the C variable with respect to the identifier. The *life-time of the C variable identifier* is the union of each life-time of each C variable with respect to the identifier.

For example, code line 2 in Figure 3 contains the definition of the C variable identifier `c`. This identifier has a scope and a life-time that is equal to the C program and the C program assertion, respectively. Hence, for each state of the C program assertion, there exists a C variable that has an rvalue of type `* int` and is associated with `c` at the state.

Regarding C variable identifiers other than those *defined* in a C program, in contrast to identifiers that are members in a C variable identifier of type `struct`, identifiers that are members in a C variable identifier of type `array`, are not defined as explained above. Instead, these identifiers are *derived* from the array identifier, e.g. `x[i]` where `x` is the array identifier. Note that derived C variable identifiers also have life-times and scopes.

Furthermore, C variables can be referenced directly in the C program by their lvalues. In such a case, the reference can be seen as an identifier with the lvalue of the referenced C variable as name – a *C variable reference identifier*. A C variable reference identifier has: a scope that is equal to the C program; and a life-time that is equal to both the C program assertion and the life-time of its referenced C variable with respect to the identifier itself. For example, in Figure 3, the address `0x40000000` is a C variable reference identifier.

A *C function* is a storage location with executable code, referenced by an lvalue. *The identifier of the C function* evaluates to the lvalue referencing the C function and is *defined* by a block of C code consisting of:

- a full *declaration* of the C function identifier including return type, name, definitions of C variable identifiers (called *formal parameters* of the C function identifier), and their order; and
- the *body* of the C function identifier, i.e. a list of declarations and statements, enclosed in brackets.

Note that this assumes that the C function identifier is neither variadic nor non-prototyped; hence, such C function identifiers are not considered in the present paper.

A C function identifier can be declared in other parts of a C program other than in the definition of the C function identifier. To differentiate between these declarations, the declaration of the C function identifier where its formal parameters are defined, will be referred to as *the* declaration of the C function identifier. Similar to C variable identifiers, C functions identifiers have scopes, which are subsets of the C program.

As an example, the code in lines 3-4 in Figure 3 constitutes the definition of the C function identifier `add`, contained in the code of `mod` where code line 3 is the declaration of `add` and where code line 4 is the body of `add`.

Despite the fact that two identifiers can have the same name, it holds that each defined C function identifier, and each defined or derived C variable identifier, can be mapped to a unique definition. Hence, two identifiers with the same name can be *distinguished* from each other. Furthermore, derived or defined identifiers are not permitted to have names that are lvalues, which means that each reference identifier is unique. In the following, when referring to *the set of identifiers in a C program*, what is really meant is the set of C function identifiers, and also the defined, derived, or C variable reference identifiers, which can be distinguished from each other; the phrase *identifiers in a C program* will be used to mean a subset of this set.

3.3 C Program Structure

A C program is typically structured into packages and layers (see e.g. AUTOSAR model [48] or Open Systems Interconnection (OSI)-layer model [49]) that encapsulate certain functionalities/services as part of the code.

For example, Figure 5 shows a structure of a C program executing on an Electric Control Unit (ECU) in a vehicle. The C program is partitioned into an application, a middleware, and a basic SW layer that are further structured into packages of C modules (each shown as a white rectangles with a folded corner) according to e.g. vehicle features (Braking, fuel estimation, etc.), communication (Controller Area Network (CAN), I/O, etc.), or Operating System (OS) services and HW interaction (Scheduling, Analogue to Digital Conversion (ADC) etc.).

The structure of the C program provides an overview and captures the overall dependencies of services [50] between different parts of the C code at different levels. For example, the application layer relies on the middleware layer to provide values that correspond to CAN-signals or sensor readings and the middleware layer expects that basic SW layer delivers voltage values of the pins of the ECU. Thus, as shown in [25, 50], organizing the specifications in parallel to the SW structure, provides a straightforward way of capturing the dependencies of services in the specifications.

A *structure of a C program* will be considered to be a rooted tree, representing an iterative partitioning of the C program where:

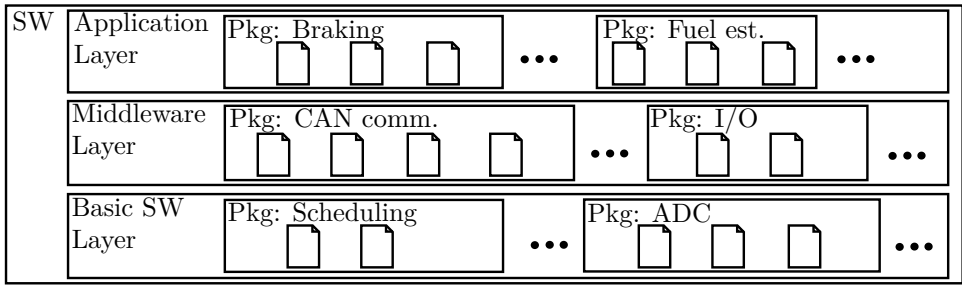


Figure 5. A structure of a C program executing on an ECU.

- the root node represents the C program;
- the translation units that constitute the C program, each represents a node in the tree; and
- the leaf nodes represent the definitions of C function identifiers in the C program and the remaining code outside of definitions of C function identifiers in the translation units, one node for each definition of a C function identifier and one for the remaining code of each translation unit.

An example of a structure of the C program `count10` is shown in Figure 2b.

Given a structure of a C program, the term *C program entity* will be used to refer to a block of code that is represented by a node in the C program structure. Furthermore, for convenience, the nodes in a C program structure will be referred to as C program entities themselves, despite the fact that they are only representations of blocks of code. Terminology from graph theory [45] will be borrowed to describe positions of C program entities, relative to each other.

4 Architecture Modeling of C Programs

This section presents the proposed mapping from non-recursive sequential C programs to architecture models. As argued in Section 1, such a mapping is needed to manage the complexity of C programs in general, and to support expressing and organizing specifications for C programs, in particular. The

architecture models are complemented with a graphical representation for practical application.

4.1 Components

This section introduces the concept of *components* to which C program entities in a C program structure are mapped. Prior to presenting the definition of a component, some preliminary mapping properties are presented.

Consider the architecture in Figure 2a, the structure of the C program shown in Figure 2b, and the mapping from C program entities in the C program structure to elements in the architecture as described in Section 2.3. It holds that the architecture is organized such that:

I) the leaf C program entities in the C program structure are mapped to the leaf elements in the architecture;

II) the parent of a C program entity in the C program structure is mapped to the parent of the element, which the C program entity is mapped to, in the architecture; and

III) the C program is mapped to the root element in the architecture.

Each of the elements in the architecture is a *component* with an interface consisting of:

IV) port variables mapped from C function and C variable identifiers in the C program; and

V) the variable of the C program, as described in Section 3.

Note that the variable of `count10` is not shown in Figure 2a since its inclusion in the interface of each component is implicit.

The architecture captures, in the manner in which the components are organized, the structure of the C program. Furthermore, as will be described in more detail in Section 4.2, the intent is that the behaviors of the components correspond to sets of sequences of evaluations of expressions involving identifiers in a C program, and the effects of the evaluations on the C program variable, i.e. the values in storage locations. Hence, the architecture of components and their interfaces provide a foundation both for organizing specifications in accordance with the structure of the C program

and for expressing specifications, e.g. in the form of contracts as described in Section 2.4, in terms of identifiers in the C program.

The definition of a component now follows.

Definition 2 (Component)**.** Given a C program and its C program variable x_{prog}, a *component* is an element $\mathbb{E} = (X, \mathsf{B})$ where X is partitioned into sets $X_1^F, \ldots, X_N^F$, X^{var}, and $\{x_{prog}\}$ such that:

a) each set $X_i^F = \{x_{i,1}, \ldots, x_{i,M_i}\} \cup \{f_i\}$ is organized as an ordered set $\mathcal{F}_i = (f_i, (x_{i,1}, \ldots, x_{i,M_i}))$, called a *function interface*; and

b) each port variable $x \in X$ is labeled as either *internal of*, *defined by*, or *external of* $\mathbb{E}$ if and only if $x \in X \setminus \{x_{prog}\}$. □

Each function interface $\mathcal{F}_i = (f_i, (x_{i,1}, \ldots, x_{i,M_i}))$ models the declaration of a C function identifier in the C program where the C function identifier is modeled by f and each of its formal parameters is modeled by a port variable $x_{i,j}$. The declaration of each C function identifier is considered to be modeled by a unique function interface. As an example of a function interface, considering the code shown in Figure 3, the declaration of `add` is modeled by a function interface $(f_{add}, (a, b, r))$ where f_{add} models `add` and its formal parameters are modeled by the port variables a, b, and r.

The port variables in $X^{var} \cup (\bigcup_{i=1}^{N} \{x_{i,1}, \ldots, x_{i,M_i}\})$ model C variable identifier in the C program, i.e. defined, derived, or reference identifiers; this includes formal parameters. Similar to the modeling of declaration of C function identifiers, each C variable identifier is considered to be modeled by a unique port variable. Unless the same name is used for two different identifiers in a C program, a port variable will have the same name as the identifier that the port variable models. For example, the port variables t and $0x40000000$ in Figure 2a model the identifiers with the same names in Figure 3.

The specifics regarding how port variables *model* identifiers will be explained in Section 4.2 and mapping properties of components where the properties specify exactly which port variables are to be included in the interface of a component, will be introduced in Section 4.3. However, assuming that the interface of a component, mapped from a C program entity, is in accordance with the properties that will be presented in Section 4.3, it holds that each variable $x \in X \setminus \{x_{prog}\}$ is labeled as:

- *internal of* $\mathbb{E}$ if x models an identifier that is defined or derived within the C program entity and the identifier has a scope that is equal to or a subset of the C program entity;
- *defined by* $\mathbb{E}$ if x models an identifier that is defined or derived within the C program entity and the identifier has a scope that includes code that is not within the C program entity; and
- *external of* $\mathbb{E}$ if x models an identifier that is neither defined nor derived within the C program entity.

For example, consider the component $\mathbb{E}_{add} = (X_{add}, \mathsf{B}_{add})$ that is shown in Figure 2a and is mapped from the definition of the C function identifier `add` in the code in Figure 3. The interface X_{add} is partitioned into two sets $\{f_{add}, a, b, r\}$ and $\{t\}$ where the former is organized as a function interface $(f_{add}, (a, b, r))$ modeling the declaration of `add`. The port variable f_{add} is labeled as *defined by* $\mathbb{E}_{add}$ due to the fact that the identifier `add` has a scope that extends beyond its own definition. The port variables a, b, and r are each labeled as *internal of* $\mathbb{E}_{add}$, which captures the fact that the C variable identifiers that the port variables model, are defined in the code of `add` and have scopes that are equal to or subsets of the definition of `add`. The port variable t is labeled as *external of* $\mathbb{E}_{add}$ since the C variable identifier that t models, is not defined in the definition of `add`.

4.2 Modeling Principles

As mentioned in Section 4.1, for an architecture of components organized in accordance with the mapping properties (I)-(V) with respect to a given C program structure, the intent is that the behaviors of the components are to be such that they correspond to sets of sequences of evaluations of expressions involving identifiers in the C program, and the effects of the evaluations on the C program variable, i.e. the values in storage locations. This section establishes a set of modeling principles such that each state in the intersection of the behaviors of the components maps to a unique value for each port variable mapped from an identifier; the *intent* that these principles hold is characterized by referring to the port variables *modeling* the identifiers. The principles are general, but will be presented in the context of the structure of the C program shown in Figure 2b and the architecture in Figure 2a.

Let $\mathsf{B}^{tot}_{count10}$ and $X^{tot}_{count10}$ be the intersection of the behaviors and the union of the interfaces of the leaf components in the architecture shown in Figure 2a, respectively. Let $\mathsf{W}_{count10}$ and $x_{count10}$ denote the assertion representing `count10` and its variable, respectively. As previously mentioned, the intent is that the behavior $\mathsf{B}^{tot}_{count10}$ is to capture the execution of the C program as evaluations of expressions and their effects on the C program variable. Hence, the C program assertion must be possible to derive from $\mathsf{B}^{tot}_{count10}$, or more specifically, it must hold that the extended projection of $\mathsf{B}^{tot}_{count10}$ onto $\{x_{count10}\}$ is equal to $\mathsf{W}_{count10}$. Note that this means that each state of $\mathsf{B}^{tot}_{count10}$ can be mapped to a state of $\mathsf{W}_{count10}$. Thus, in accordance with Section 3.1, each initial state of $\mathsf{B}^{tot}_{count10}$ maps to an invocation of `main`, and each non-initial state of $\mathsf{B}^{tot}_{count10}$ maps to an evaluation of an expression in `count10`. Furthermore, it holds that the life-times of identifiers and C variables with respect to identifiers, can also be expressed as sequences of $\mathsf{B}^{tot}_{count10}$.

In the following, let each state ω' of $\mathsf{B}^{tot}_{count10}$ be referred to as

- *a call to a C function identifier* `foo` if ω' maps to an evaluation of an expression where the evaluated value is the lvalue that also `foo` evaluates to;
- *a return to a C function identifier* `foo` if ω' maps to an evaluation of an expression that is within the definition of `foo` and it holds that there exists a transition (ω, ω') of $\mathsf{B}^{tot}_{count10}$ such that ω is not a call to `foo` and ω does not map to an evaluation of an expression that is within the definition of `foo`.

Note that these concepts trivially extend to the C program assertion $\mathsf{W}_{count10}$ and also any C program assertion.

As an example of a call to a C function identifier, consider a state ω of $\mathsf{B}^{tot}_{count10}$ where ω maps to an evaluation of the expression `step`, represented by a leaf node in Figure 4. Since the state ω maps to an evaluation of `step`, ω is a call to `step`, which will trigger the execution of code in the rvalue of the C function with an lvalue that `step` evaluates to. Hence, in accordance with Section 3.1, the state ω specifies a point right before executing code in the body of `add`, contained in the code line 6 in Figure 3. As an example of a return to a C function identifier, a state of $\mathsf{B}^{tot}_{count10}$ where this state maps to an evaluation of the expression `*r=res` in the code line 4 in Figure 3, is a return to `step`.

4.2.1 C Function Identifiers Modeled by Port Variables

Consider a variable f in a function interface $(f, (x_1, \ldots, x_N))$ modeling the declaration of a C function identifier `foo`. The port variable f is a pair (f^e, f^r) of variables where f^r *models* the passing of return values and f^e *models* the execution status of `foo`, e.g. whether code within its definition is executing or not. This section will describe this modeling in detail. Observe that it will be assumed that preemption of a C function identifier, due to an interrupt, does not occur; how preemption can be modeled is instead discussed in Section 4.2.4.

Suppose that $f \in X^{tot}_{count10}$. In the case where `foo` is not `main`, for each transition (ω, ω') of $\mathsf{B}^{tot}_{count10}$ where ω is a return of `foo`, the intent is that it holds that $f^r(\omega)$ is equal to the value of the expression evaluation that ω' maps to; note that the evaluated value is possibly *void*. In any other state ω'' of $\mathsf{B}^{tot}_{count10}$, the intent is that it holds that $f^r(\omega'')$ is equal to a value *nil*, which captures the fact that no return value is passed. On the other hand, if `foo` is `main`, then the intent is that $f^r(\omega)$ is equal to nil for each non-terminating state ω of $\mathsf{B}^{tot}_{count10}$. For each terminating state ω' of $\mathsf{B}^{tot}_{count10}$, the intent is that it holds that $f^r(\omega')$ is equal to the value of the expression evaluation that ω' maps to if `main` is not of type `void`, and $f^r(\omega') = void$, otherwise.

For example, let (ω, ω') be a transition of $\mathsf{B}^{tot}_{count10}$ where ω and ω' map to evaluations of the expressions `*r=res` and `add(t,1,@t)` in the code line 4 and 6 in Figure 3, respectively. As previously presented in Section 4.1, the declaration of `add` is modeled by the function interface $(f_{add}, (a, b, r))$. In accordance with the manner in which f^r_{add} models `add` as presented above, the intent is that $f^r_{add}(\omega) = void$ and $f^r_{add}(\omega') = nil$. As a second example, let $(f_{main}, ())$ be the function interface modeling `main` and ω'' be a terminating state of $\mathsf{B}^{tot}_{count10}$ where ω'' is mapped to an evaluation of the constant 0, represented as a leaf node in Figure 4. In accordance with the presented principles, it follows that $f^r_{main}(\omega'') = 0$.

Regarding the variable f^e that *models* the execution status of `foo`, it takes values from the set $\{0, 1, 2\}$. Formulated in the context of a system executing `count10`, considering the C function with an lvalue that `foo` evaluates to, the intent is that it holds that the value of f^e is:

2 – if the C function is on the call stack and code in its rvalue is not executing;

1 – if the C function is on the call stack and code in its rvalue is executing; and

0 – if the C function is not on the call stack.

Formulated in terms of $\mathsf{B}^{tot}_{count10}$, the intent is that it holds that for each non-initial state ω' of B, the value of f^e is:

2 – if the state ω' either:

a) maps to an evaluation of an expression that is within in the definition of `foo` and ω' is a call to a C function identifier; or

b) does not map to an evaluation of an expression that is within the definition of `foo` and ω' is neither a call nor a return to `foo` and it holds that there exists a sequence $(\omega, \omega_1, \ldots, \omega_{M\geq 0}, \omega')$ of $\mathsf{B}^{tot}_{count10}$ where

- the state ω maps to an evaluation of an expression that is within the definition of `foo` and ω is a call to a C function identifier, and
- each ω_i is not a return to `foo`.

1 – if the state ω' either

a) maps to an evaluation of an expression that is within the definition of `foo` and ω' is neither a call nor a return to a C function identifier, or

b) is a call or a return to `foo`; and

0 – Otherwise.

In accordance with Section 3, each initial state ω of $\mathsf{B}^{tot}_{count10}$ is an invocation of `main`. Hence, for each initial state ω of $\mathsf{B}^{tot}_{count10}$, the intent is that it holds that $f^e(\omega) = 1$ if `foo` is `main` and $f^e(\omega) = 0$, otherwise.

For example, consider port variables f_{add}, f_{step}, and f_{main} modeling the respective C function identifiers `add`, `step`, and `main` and a state ω of $\mathsf{B}^{tot}_{count10}$ that maps to an evaluation of the expression `*r=res` in the code line 4 in Figure 3. The state ω is a return to `step`; hence, the intent is that $f^e_{step}(\omega) = 1$ in accordance with the condition (1-b). Furthermore, the intent is that it holds that $f^e_{add}(\omega) = 0$ due to the fact that ω maps to an

evaluation of an expression that is within the definition of `add` and ω is a return to `step`. In accordance with (2-b), the intent is also that it holds that $f^e_{main}(\omega) = 2$ since the C function with an lvalue that `main` evaluates to, is on the call stack and code in the rvalue of the C function is not executing.

As a second example, consider a state ω' of $\mathsf{B}^{tot}_{count10}$ where ω' is mapped to an evaluation of the expression `c*<10` that is in the code line 10 in Figure 3 and represented as a node in the AST in Figure 4. In accordance with the condition (1-a), the intent is that it holds that $f^e_{main} = 1$. Moreover, the intent is that $f^e_{add}(\omega) = 0$ and $f^e_{main}(\omega) = 0$.

4.2.2 C Variable Identifiers Modeled by Port Variables

This section describes how port variables in $X^{tot}_{count10}$ model C variable identifiers. The case where a C variable identifier is not a pointer is first described, followed by the case where the identifier is a pointer.

Consider a port variable $x \in X^{tot}_{count10}$ that models a C variable identifier `var` that is not a pointer. For each state ω of $\mathsf{B}^{tot}_{count10}$ where a C variable is associated with `var`, the intent is that it holds that $x(\omega)$ is equal to the rvalue of the C variable. For example, for each state ω' of $\mathsf{B}^{tot}_{count10}$, the intent is that $0x40000000(\omega')$ is equal to the rvalue of the C variable that has the lvalue $0x40000000$.

In the case where `var` is a pointer and a C variable is associated with `var` at ω, the intent is that the rvalue of the C variable is either $NULL$ or an lvalue of another C variable. The same conditions as in the case where `var` is not a pointer apply, except when there exists another identifier `var2` modeled by a port variable $x \in X^{tot}_{count10}$ such that the rvalue of the C variable is an lvalue of another C variable that is associated with `var2` at ω. To capture this dependency between the identifiers `var` and `var2`, the intent is that $x(\omega)$ is equal to the *name* of the port variable, instead of the lvalue. For example, in each state ω' of the life-time of the C variable identifier that is modeled by the port variable t and defined in the code line 6 in Figure 3, the intent is that $t(\omega')$ is equal to the name of the port variable c.

Regardless if `var` is a pointer or not, the intent is that it holds that $x(\omega) = nil$ if ω is not in the life-time of `var`. For example, in any initial state ω_0 of $\mathsf{B}^{tot}_{count10}$, the intent is that it holds that $a(\omega_0)$, $b(\omega_0)$, and $r(\omega_0)$ are equal to nil.

4.2.3 Behavior and Interface of a Single Component

Section 4.2.1 and Section 4.2.2 described how the port variables in $X^{tot}_{count10} \setminus \{x_{prog}\}$ model identifiers in `count10`. Notably, this was achieved by describing the manner in which the *intersection* of the component behaviors $\mathsf{B}^{tot}_{count10}$ is intended to constrain $X^{tot}_{count10}$.

Concerning the behavior of a *single* component mapped from a C program entity, the intent is that the behavior is to capture the constraints that the C program entity has on the rest of the C program whenever code within the C program entity is executing. Notably, in accordance with Section 2.3, an element can only constrain port variables in its interface; thus, the interface of the component must be such that this is possible. Therefore, Section 4.3 will establish properties of component interfaces such that the interfaces are indeed sufficient for capturing such constraints.

4.2.4 Discussion

This section discusses certain aspects mentioned in Sections 4.2.1-4.2.3 in more depth.

Modeling Preemption As mentioned in Section 4.2.1, preemption due to an interrupt was not considered when describing how port variables model C function identifiers. The following will describe how preemption can be captured. Assuming that C functions identifiers may preempt other C functions identifiers can be identified in the C program and that such preemptive C function identifiers are only invoked through interrupts, a state ω' of a C program assertion is *a preemption of a C function identifier* `foo` if:

- ω' maps to an expression that is within a definition of a preemptive C function identifier; and
- there exists a transition (ω, ω') of the C program assertion such that ω maps to an expression within the definition of `foo`.

Note that in accordance with Section 4.2.1, a preemption of `foo` is a return to `foo`. Hence, to also capture preemption, these cases must be distinguished from each other since the intent is that the port variable modeling the return value of `foo`, shall have the value *nil* in case the state is a preemption of `foo` and not *nil*, otherwise.

Constraints Captured by Component Behaviors In Section 4.2.3, it was stated that the intent is that the behavior of a leaf component, mapped from a C program entity, is to capture the constraints that the C program entity has on the rest of the C program whenever code within the C program entity is executing. While this is true, it also holds that the behavior must capture other constraints in order for the modeling principles in Sections 4.2.1-4.2.2 to hold. Considering a system executing the C program, these other constraints are actually properties of the processor and memory. Examples of such properties are: allocation/deallocation of memory, which means constraining the value of a port variable, modeling a C variable identifier, to either have the value *nil* or not have the value *nil*; and transferring control back to a caller, which means constraining a port variable $f = (f^e, f^r)$ such that $f^e(\omega) = 2$ and $f^e(\omega) = 1$ for a transition (ω, ω') of a C program assertion.

As another example, in Figure 2a, consider port variable c modeling the C variable identifier with the same name. As can be seen in the code in Figure 3, this C variable identifier does not have any direct dependencies to the execution of code within the definition of `add`. In accordance with the modeling principles in Sections 4.2.1-4.2.2, it holds that during sequences where $f_{add}(\omega) = 1$ for each state ω in the sequence, the value of the port variable c should be constant. In accordance with Section 2.3, since c is not a port variable of $\mathbb{E}_{add}$, its behavior cannot enforce such a constraint. This means that the behavior of another leaf component, e.g. $\mathbb{E}_{step}$, must enforce this. Notably, it does not really matter which component enforces such a constraint since, as previously mentioned, such a constraint will not be enforced by the execution of code, but is rather a property of the processor and memory.

4.3 Representing and Modeling a C Program Structure as Architecture of Components

Section 4.1 introduced the concept of components to which C program entities in a C program structure are mapped. Furthermore, properties of the interfaces of components were presented where the properties expressed the labels of port variables mapped from C function and C variable identifiers, respectively, and Section 4.2 presented principles for how port variables model identifiers. However, these properties did not establish a unique mapping from a C program entity to a component since the properties did not

express the set of port variables that is to be included in the interface of the component, and how this interface shall be partitioned into the subsets that are organized as function interfaces.

Therefore, this section introduces additional properties of component interfaces and their organization, considering the C program entities that the components are mapped from. In essence, the properties ensure that the interface of a component mapped from a C program entity, consists of the port variables modeling identifiers that are either: derived or defined within the C program entity and have no dependencies to execution of code *outside* of the C program entity; or not derived or defined within the C program entity and has dependencies to execution of code *within* the C program entity. That is, the interface captures an encapsulation of the C program entity; it only exposes the external dependencies that the C program entity has to the rest of the C program in terms of identifiers, and vice versa.

The properties will be introduced in the context of the structure of the C program shown in Figure 2b and the architecture in Figure 2a. In addition to these properties, this section also describes a graphical syntax used for representing an architecture of components. The graphical syntax is first presented in Section 4.3.1, followed by Section 4.3.2 and Section 4.3.3 that present properties of interfaces of leaf components and non-leaf components in an architecture such as the one in Figure 2a, respectively. Section 4.4 will then present the complete set of properties of components of an architecture that models a C program structure.

4.3.1 Graphical Syntax

The graphical syntax is summarized in Figure 6. As can be seen, a symbol attached to an edge of a box representing a port variable, both represents the type of the identifier modeled by the port variable and the label of the port variable. A triangle and a diamond both represents the fact that the port variable models a C variable identifier; a circle represents that the port variable models a C function identifier. Boxes representing the port variables that constitute a function interface, are enclosed in dashed brackets.

Notably, as can be seen in Figure 2a, several symbols can be attached to different edges of the same box, see e.g. the box that represents c and is on an edge of each of the rectangles representing $\mathbb{E}_{main}$, $\mathbb{E}_{mod2}$, and $\mathbb{E}_{mod}$. In relation to a rectangle with an edge to which a box representing a port variable is attached, the specific edge to which a symbol is attached to the

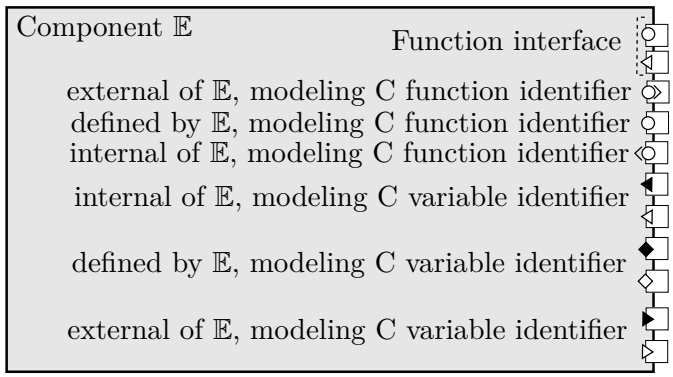


Figure 6. Graphical Syntax.

box, represents the label of the port variable with respect to the component representing the rectangle. More specifically, if a symbol is attached to an edge of a box where the edge is within a rectangle representing a component, then that symbol represents the specific label of the port variable with respect to that component. That is, each of the symbols attached to boxes in Figure 6 represents the labels of port variables with respect to the component $\mathbb{E}$. A triangle, or a circle with a hat, pointing inwards and outwards from the center of the rectangle, represent that the port variable is labeled as internal and external of the component, respectively. A diamond, or a circle without a hat, represents that the port variable is labeled as defined by the component.

For example, as shown in Figure 2a, the triangles attached to the box that represents c and is on edges of the rectangles representing $\mathbb{E}_{main}$, $\mathbb{E}_{mod2}$, and $\mathbb{E}_{mod}$, represent labels of c with respect to the components $\mathbb{E}_{main}$, $\mathbb{E}_{mod2}$, and $\mathbb{E}_{mod}$. The triangles pointing outwards from the center of the rectangles representing $\mathbb{E}_{main}$ and $\mathbb{E}_{mod2}$, represents that c is labeled as external of both $\mathbb{E}_{main}$ and $\mathbb{E}_{mod2}$; the triangle pointing outwards from the center of the rectangle representing $\mathbb{E}_{mod}$, represents that c is labeled as external of $\mathbb{E}_{mod}$.

Furthermore, if a port variable models a C variable identifier, then a hollow symbol represents the fact that the C variable identifier has a life-time that is not equal to the C program assertion; a solid symbol represents that the C variable identifier has a life-time that is equal to the C program assertion. Hence, as shown in Figure 2a, while the symbols attached to

boxes representing c and $0x40000000$ are solid, the symbols attached to boxes representing port variables a, b, r, and t, are hollow.

4.3.2 Properties of Leaf Component Interfaces

As discussed in Section 4.2.3, for each component in $\mathscr{A}$ where the component is mapped from a C program entity, the intent is that the interface of the component is such that it is possible for the behavior to express the constraints that the C program entity has on the rest of the C program whenever code within the C program entity is executing. Considering the previously mentioned intent of the interface capturing an encapsulation of the C program entity, this means that identifiers can only be encapsulated if they are derived or defined within a C program entity, and has no dependencies to the execution of code outside of the C program entity.

For example, consider the definition of the C function identifier `main` and the component $\mathbb{E}_{main} = (X_{main}, \mathsf{B}_{main})$ in Figure 2a. The declaration of `main` is mapped to a function interface $(f_{main}, ())$. Let (ω, ω') be a transition of $\mathsf{B}^{tot}_{count10}$ where $f_{main}(\omega) = (1, nil)$, $f_{main}(\omega') = (2, nil)$, and $f_{step}(\omega') = (1, nil)$. In accordance with the manner in which f_{main} and f_{step} model the identifiers `main` and `step` as described in Section 4.1, the state ω' is a call to the identifier `step`. Notably, considering that $f_{main}(\omega) = (1, nil)$, it is the code *within* the definition of `main` that triggers the execution of the compiled code in the rvalue of the C function with an lvalue that `step` evaluates to. To capture this, B_{main} must constrain both f_{main} and f_{step} as indicated above. Therefore, in accordance with Section 2.3, the interface of $\mathbb{E}_{main}$ must also contain both f_{main} and f_{step}. These insights will now be generalized as follows.

Let F and V be the set of C function identifiers and the set of C variable identifiers in `count10`, respectively. Furthermore, let $\mathscr{A}$ denote the set of components in the architecture shown in Figure 2a. For each component $\mathbb{E} = (X, \mathsf{B}) \in \mathscr{A}$ mapped from a definition of a C function identifier, where X is partitioned into sets $X^F_1, \ldots, X^F_N$, X_{var}, and $\{x_{count10}\}$ such that each set X^F_i is organized as a function interface $\mathcal{F}_i$, it holds that:

- for each C function identifier `foo` $\in F$, there exists a function interface $\mathcal{F}_i$ modeling the declaration of `foo` if and only if `foo` is either
  - the C function identifier itself, or

– there exists a state of $\mathsf{W}_{count10}$ where this state is a call to `foo` and maps to an evaluation of an expression within the definition of the C function identifier.

Thus, it holds that subsets of the interface of $\mathbb{E}_{main}$ are organized as the function interfaces $(f_{main}, ())$ and $(f_{step}, ())$. Furthermore, in accordance with Section 4.1 and as shown in Figure 2a, f_{main} is labeled as defined by $\mathbb{E}_{main}$ since $(f_{main}, ())$ models the declaration of `main` and the definition of `main` is mapped to $\mathbb{E}_{main}$. In contrast, f_{step} is labeled as external of $\mathbb{E}_{main}$ since `step` is not defined in the definition of `main`.

Regarding the component $\mathbb{E}_{add}$ as previously introduced, there does not exist a state of $\mathsf{W}_{count10}$ where this state is mapped to an evaluation of an expression within the definition of `add` and where this state is a call to a C function identifier. Thus, as shown in Figure 2a, only a single subset of the interface of $\mathbb{E}_{add}$ is organized as a function interface, namely the set $\{f_{add}, a, b, r\}$ organized as the function interface $(f_{add}, (a, b, r))$ modeling the declaration of `add`.

Considering the component $\mathbb{E}_{step}$ mapped from the definition of `step`, the set $\{f_{step}\}$ is organized as a function interface $(f_{step}, ())$ modeling the declaration of `step`, and thus, f_{step} is labeled as defined by $\mathbb{E}_{step}$. In addition, since there exists a state of $\mathsf{W}_{count10}$ where this state is mapped to an evaluation of an expression in the definition of `step` and where this state is a call to the C function identifier `add`, it holds that a subset of the interface of $\mathbb{E}_{step}$ is organized as the function interface $(f_{add}, (a, b, r))$. Due to the fact that neither `add` nor the C variable identifiers modeled by the port variables a, b, and r, are defined in the definition of `step`, it holds that f_{add}, a, b, and r are each labeled as external of $\mathbb{E}_{step}$ as shown in Figure 2a.

Now that properties, expressing the organization of subsets of the interface of a component into function interfaces, have been presented, properties that express the modeling of C variable identifiers by port variables in the interface, will be introduced through two examples that now follow.

As the first example, consider the component $\mathbb{E}_{add}$ and the expression `*r=res` in the code line 4 in Figure 3. Notably, for each state ω of $\mathsf{B}^{tot}_{count10}$ where ω maps to an evaluation of `*r=res`, it holds that a C variable is assigned with the evaluated value res where this C variable is associated with the identifier modeled by t at the state ω. This means, in accordance with the modeling principles in Section 4.2, that $t(\omega)$ must be equal to the value res. Hence, in order for B_{add} to express the constraints that the

definition of `add` has on the rest of the C program whenever code within the definition of `add` is executing, B_{add} must constrain the port variable t.

As the second example, consider the component $\mathbb{E}_{main}$ and the code line 10 in Figure 3. Let (ω, ω') be a transition of $\mathsf{B}^{tot}_{count10}$ where $f^{e}_{main}(\omega) = 1$ and $f^{e}_{main}(\omega') \neq 1$. In accordance with the manner in which f_{main} models the identifier `main` as described in Section 4.1, the state ω' either maps to an evaluation of the identifier `step` in the function call expression `step()` or to an evaluation of the constant `0` in the return statement `return 0;` i.e. it either holds that $f_{main}(\omega') = (2, nil)$ or $f_{main}(\omega') = (0, 0)$. Notably, as can be seen in code line 10 of Figure 3 and in the AST in Figure 4, whether the function call expression `step()` or the return statement `return 0;` is executed, depends on if the expression `*c<10` is evaluated to true or not. In turn, the expression `*c` must be evaluated in order to evaluate `*c<10`. The expression `*c` evaluates to the rvalue of a C variable with the lvalue $0x40000000$, which is also the rvalue of a C variable that is associated with the C variable identifier modeled by c at state ω. Hence, to determine whether it holds that $f_{main}(\omega') = (2, nil)$ and $f^{e}_{step}(\omega') = 1$, or $f_{main}(\omega') = (0, 0)$, both the rvalues of the C variables that are associated with the C variable identifiers modeled by c and $0x40000000$ at ω, are needed. For example, it can be determined that $f_{main}(\omega') = (0, 0)$ if $0x40000000(\omega) \geq 10$ and $t(\omega) = 0x40000000$, but not if only one of these values are known. Thus, the behavior B_{main} must constrain both the port variables c and $0x40000000$.

Hence, in the first and second example, it holds that the port variable t and the port variables c and $0x40000000$ must be in the interface of $\mathbb{E}_{add}$ and $\mathbb{E}_{main}$, respectively. These insights will now generalized.

In general, for each component $\mathbb{E} = (X, \mathsf{B}) \in \mathscr{A}$ mapped from a definition of a C function identifier, where X is partitioned into sets $X^{F}_{1}, \ldots, X^{F}_{N}$, X_{var}, and $\{x_{count10}\}$ such that each set X^{F}_{i} is organized as a function interface $\mathcal{F}_i$, it holds that:

- for each C variable identifier `var` $\in V$, there exists a variable $x \in X$ modeling `var` if and only if `var` is either:

  i) defined or derived within the definition of the C function identifier and there exists a state ω of $\mathsf{W}_{count10}$ such that ω maps to an evaluation of an expression where

  - the expression is not a function call expression and is not within the definition of the C function identifier, and

- the lvalue or rvalue of a C variable that is associated with `var` at state ω, is used for evaluating the expression;

ii) not defined or derived within the definition of the C function identifier and there exists a state ω of $\mathsf{W}_{count10}$ where ω maps to an evaluation of an expression where

- the expression is not a function call expression and is within the definition of the C function identifier, and
- the lvalue or rvalue of a C variable that is associated with `var` at state ω, is used for evaluating the expression; or

iii) a formal parameter of a C function identifier with a declaration modeled by a function interface $\mathcal{F}_i$.

Considering the two previously presented examples regarding $\mathbb{E}_{add}$ and $\mathbb{E}_{main}$, the property (i) and (ii) enforce the inclusion of the port variable t and the port variables c and $0x40000000$ in the interfaces of $\mathbb{E}_{add}$ and $\mathbb{E}_{main}$, respectively. The property (iii) is included for the sake of completeness.

In accordance with Section 4.1 and as shown in Figure 2a, since the C variable identifier, modeled by the port variable t, is not defined or derived in the definition of `add`, it holds that port variable t is labeled as external of $\mathbb{E}_{add}$. Considering the interface of $\mathbb{E}_{add}$, it can be noted that the C variable identifier `res` is not modeled as a port variable of $\mathbb{E}_{add}$. This is due to the fact that this C variable identifier does not have any dependencies to the execution of code outside of the definition of `add`.

Regarding the labeling of the port variables c and $0x40000000$ with respect to $\mathbb{E}_{main}$, none of the C variable identifiers modeled by the port variables c and $0x40000000$, are defined or derived in the definition of `main`. Hence, both the port variables c and $0x40000000$ are labeled as external of $\mathbb{E}_{main}$. Notably, in accordance with the property (ii) and as shown in Figure 2a, the port variables c and $0x40000000$ are also on the interface of $\mathbb{E}_{step}$. Furthermore, both these port variables are labeled as external of $\mathbb{E}_{step}$.

Now that the properties of the interfaces of the leaf components $\mathbb{E}_{add}$, $\mathbb{E}_{add}$, and $\mathbb{E}_{main}$ have been explained in thorough, properties of the interfaces of the components $\mathbb{E}_{c2}$ and $\mathbb{E}_{c8}$ follow. The components $\mathbb{E}_{c2}$ and $\mathbb{E}_{c8}$ are mapped from the code `int * c = (int *) 0x40000000;` and also the code `extern int * c;` in Figure 3, respectively.

In general, for each component $\mathbb{E} = (X, \mathsf{B}) \in \mathscr{A}$ mapped from a leaf C program entity that is not a definition of a C function identifier, where X is

partitioned into sets $X_1^F, \ldots, X_N^F$, X_{var}, and $\{x_{count10}\}$ such that each set X_i^F is organized as a function interface $\mathcal{F}_i$, it holds that

- $X_{var} \cup \{x_{count10}\} = X$; and
- for each C variable identifier `var` $\in V$, there exists a variable $x \in X$ modeling `var` if and only if either:
  - `var` is defined or derived within the C program entity, or
  - the lvalue or rvalue of a C variable that is associated with `var` at an initial state of $\mathsf{W}_{count10}$, is used for evaluating an expression within the C program entity;

The property that $X_{var} \cup \{x_{prog}\} = X$ expresses that port variables of leaf components that are not mapped from a definition of a C function identifier, do not model C function identifiers – only C variable identifiers. Since the code `int * c = (int *) 0x40000000;` constitutes a definition of the C variable identifier modeled by the port variable c, it holds that c is labeled as defined by $\mathbb{E}_{c2}$. Furthermore, consider that the lvalue of a C variable is associated with the reference C variable identifier modeled by the port variable $0x40000000$ at an initial state of W_{prog}. Since this C variable is used for evaluating the initializer expression `(int *) 0x40000000`, it holds that the port variable $0x40000000$ is also in the interface of $\mathbb{E}_{c2}$. The port variable $0x40000000$ is labeled as external of $\mathbb{E}_{c2}$ since it is not defined in the code `int * c = (int *) 0x40000000;`. The interface of $\mathbb{E}_{c8}$ is empty since no C variable identifiers are defined in the code `extern int * c;` that is mapped to $\mathbb{E}_{c8}$.

4.3.3 Properties of Non-Leaf Component Interfaces

This section introduces properties of the interfaces of non-leaf components in an architecture such as the one shown in Figure 2a. As previously mentioned in Section 4.3.2, the set $\mathscr{A}$ denote the set of components in the architecture shown in Figure 2a and the sets F and V denote the sets of C function identifiers and C variable identifiers in `count10`, respectively.

Consider the interfaces of the components $\mathbb{E}_{mod}$ and $\mathbb{E}_{mod2}$ mapped from the translation units `mod` and `mod2`, and the component $\mathbb{E}_{count10}$ mapped from the C program `count10`. In general, for each component $\mathbb{E} = (X, \mathsf{B}) \in \mathscr{A}$ mapped from a non-leaf C program entity, where X is partitioned into

sets $X_1^F, \ldots, X_N^F$, X_{var}, and $\{x_{count10}\}$ such that each set X_i^F is organized as a function interface $\mathcal{F}_i$, it holds that:

- for each C function identifier `foo` $\in F$, there exists a function interface $\mathcal{F}_i = (f_i, (x_{i,1}, \ldots, x_{i,N_i}))$ modeling the declaration of `foo` if and only if the definition of `foo` is either:
  - both a descendant of the C program entity and the definition of the C function identifier `main`;
  - a descendant of the C program entity and there exists a state of $\mathsf{W}_{count10}$ where this state is a call to `foo` and maps to an evaluation of an expression that is not within the C program entity; or
  - a non-descendant of the C program entity and there exists a state of $\mathsf{W}_{count10}$ where this state is a call to `foo` and maps to an expression that is within the C program entity;
- for each C variable identifier `var` $\in V$, there exists a port variable $x \in X$ modeling `var` if and only if `var` is either:
  - a formal parameter of a C function identifier with a declaration modeled by a function interface $\mathcal{F}_i$;
  - defined or derived within the C program entity and there exists a state ω of $\mathsf{W}_{count10}$ such that ω maps to an evaluation of an expression where
    - the expression is not a function call expression and is not within the C program entity, and
    - the lvalue or rvalue of a C variable that is associated with `var` at state ω, is used for evaluating the expression; or
  - not defined or derived within the C program entity and there exists a state ω of $\mathsf{W}_{count10}$ where ω maps to an evaluation of an expression where
    - the expression is not a function call expression and is within the C program entity, and
    - the lvalue or rvalue of a C variable that is associated with `var` at state ω, is used for evaluating the expression.

In accordance with the properties above, except for the port variable f_{main}, it holds that the only port variables on the interfaces of $\mathbb{E}_{mod}$ and $\mathbb{E}_{mod2}$ are port variables that are in the interface of a child of both $\mathbb{E}_{mod}$ and $\mathbb{E}_{mod2}$. That is, f_{main} is the only port variable that could be removed from the interfaces of $\mathbb{E}_{mod}$ and $\mathbb{E}_{mod2}$ without violating the condition (b) of Definition 1, i.e. the definition of an architecture. The reason why f_{main} is a port variable of $\mathbb{E}_{mod2}$, and also of $\mathbb{E}_{count10}$, is that `main` is invoked from outside of the C program, and hence, has a dependency outside of the C program. Notably, $0x40000000$ is also a port variable of $\mathbb{E}_{count10}$. This is due to the fact that the lvalue $0x40000000$ may be memory-mapped to a peripheral device, e.g. a sensor, which can access or modify the rvalue of the C variable with $0x40000000$ as lvalue.

4.3.4 Discussion

Sections 4.3.2 and 4.3.3 introduced properties expressing *exactly* the port variables that are to be included in the interface of each component in an architecture organized in accordance with the mapping properties (I)-(V) with respect to a given C program structure. As will be shown in Section 4.4.2, this means that the interfaces of the components are unique for a given C program structure. However, there are cases where it is relevant to include other port variables than those explicitly specified to be included by the introduced properties. An example of such a case is when it is infeasible – in practice – to identify whether a certain C variable identifier has dependencies to a C program entity or not, due to e.g. pointer dereferencing in several levels or complex control flows. In such a case, the interface would need to be over-approximated by including the port variable modeling the C variable identifier, regardless if it has dependencies to the C program entity or not.

As another example, consider the fact that a subset of the interface of the root component in an architecture such as the one shown in Figure 2a, is organized as a function interface modeling the declaration of `main`. As previously explained, this is due to the fact that `main` is invoked from outside of the C program. Notably, in addition to `main`, there may exist other C function identifiers that can be invoked from outside of the C program – namely Interrupt Service Routines (ISRs). In such cases, similar to the function interface modeling the declaration of `main`, port variables modeling ISR identifiers and their formal parameters should be included in the

interface of the root component; however, the properties in Sections 4.3.2 and 4.3.3 do not ensure that this is the case. This is due to the fact that ISR functionality is not a built-in feature of the C language; rather, it is established by using compiler-specific directives, of which in-depth analyzes are considered to be out of scope of this paper.

Section 4.4.3 will describe how the cases where additional port variables need to be included in component interfaces, can be managed.

4.4 Modeling C Program Structure as Architecture of Components

This section first summarizes the properties of a set of components mapped from a given C program structure, as presented in Sections 4.1-4.3. It will then be shown that the structural conditions of an architecture as presented in Definition 1 hold for a set of components with such properties. Subsequently, it will be shown that an architecture of components with such properties, called *an architecture model of the C program structure*, is unique for the given C program structure with respect to the interfaces of the components in the architecture.

4.4.1 Properties of Components Mapped From C Program Structure

Consider a structure of a sequential non-recursive C program as described in Section 3 where:

- x_{prog} and W_{prog} are the C program assertion and C program variable, respectively, as introduced in Section 3.1;
- F and V are the sets of C function and C variable identifiers in the C program, respectively, as described in Section 3.2.

In accordance with Sections 4.1 and 4.2, let $X_{F\cup V} \subseteq \Xi$ be a set such that there is a bijective mapping from $F \cup V$ to $X_{F\cup V}$ where each variable $x \in X_{F\cup V}$ is said to *model* the identifier that it is mapped from.

Let $\mathscr{A}$ be a set of components, such that:

I) in accordance with Section 4.1, the components in $\mathscr{A}$ are organized as a rooted tree where:

a) the leaf C program entities in the C program structure are mapped to the leaf components of $\mathscr{A}$;

b) the parent of a C program entity in the C program structure is a mapped to the parent of the component that the C program entity is mapped to in $\mathscr{A}$; and

c) the C program is mapped to the root component of $\mathscr{A}$;

II) for each component $\mathbb{E} = (X, \mathsf{B})$ in $\mathscr{A}$, where a C program entity is mapped to $\mathbb{E}$ and where, in accordance with Definition 2, X is partitioned into sets $X_1^F, \ldots, X_N^F$, X_{var}, and $\{x_{prog}\}$ such that each set X_i^F is organized as a function interface $\mathcal{F}_i$, it holds that:

a) $X \setminus \{x_{prog}\} \subseteq X_{F \cup V}$;

b) in accordance with Section 4.1, each variable $x \in X \setminus \{x_{prog}\}$ is labeled as:

i) internal of $\mathbb{E}$ if x models an identifier that is defined or derived within the C program entity and the identifier has a scope that is equal to or a subset of the C program entity;

ii) defined by $\mathbb{E}$ if x models an identifier that is defined or derived within the C program entity and the identifier has a scope that includes code that is not within the C program entity; and

iii) external of $\mathbb{E}$ if x models an identifier that is neither defined nor derived within the C program entity;

c) in accordance with Section 4.3.2, if the C program entity is a leaf C program entity, which is not a definition of a C function identifier, then the component $\mathbb{E} = (X, \mathsf{B})$ is such that:

i) it holds that $X_{var} \cup \{x_{prog}\} = X$; and

ii) for each C variable identifier `var` $\in V$, there exists a variable $x \in X$ modeling `var` if and only if either

- `var` is defined or derived within the C program entity, or
- the lvalue or rvalue of a C variable that is associated with `var` at an initial state of W_{prog}, is used for evaluating an expression within the C program entity;

d) in accordance with Section 4.3.2, if the C program entity is a definition of a C function identifier, then the component $\mathbb{E} = (X, \mathsf{B})$ is such that:

   i) for each C function identifier `foo` $\in F$, there exists a function interface $\mathcal{F}_i$ modeling the declaration of `foo` if and only if `foo` is either
      - the C function identifier itself, or
      - there exists a state of W_{prog} where this state is a call to `foo` and maps to an evaluation of an expression within the definition of the C function identifier; and

   ii) for each C variable identifier `var` $\in V$, there exists a variable $x \in X$ modeling `var` if and only if `var` is either:
      - a formal parameter of a C function identifier with a declaration modeled by a function interface $\mathcal{F}_i$;
      - defined or derived within the definition of the C function identifier and there exists a state ω of W_{prog} such that ω maps to an evaluation of an expression where
        – the expression is not a function call expression and is not within the definition of the C function identifier, and
        – the lvalue or rvalue of a C variable that is associated with `var` at state ω, is used for evaluating the expression; or
      - not defined or derived within the definition of the C function identifier and there exists a state ω of W_{prog} where ω maps to an evaluation of an expression where
        – the expression is not a function call expression and is within the definition of the C function identifier, and
        – the lvalue or rvalue of a C variable that is associated with `var` at state ω, is used for evaluating the expression; and

e) in accordance with Section 4.3.3, if the C program entity is a non-leaf C program entity, then the component $\mathbb{E} = (X, \mathsf{B})$ is such that:

i) for each C function identifier `foo` $\in F$, there exists a function interface $\mathcal{F}_i = (f_i, (x_{i,1}, \ldots, x_{i,N_i}))$ modeling the declaration of `foo` if and only if the definition of `foo` is either:

- both a descendant of the C program entity and the definition of the C function identifier `main`;
- a descendant of the C program entity and there exists a state of W_{prog} where this state is a call to `foo` and maps to an evaluation of an expression that is not within the C program entity; or
- a non-descendant of the C program entity and there exists a state of W_{prog} where this state is a call to `foo` and maps to an expression that is within the C program entity;

ii) for each C variable identifier `var` $\in V$, there exists a port variable $x \in X$ modeling `var` if and only if `var` is either:

- a formal parameter of a C function identifier with a declaration modeled by a function interface $\mathcal{F}_i$;
- defined or derived within the C program entity and there exists a state ω of W_{prog} such that ω maps to an evaluation of an expression where
  - the expression is not a function call expression and not within the C program entity, and
  - the lvalue or rvalue of a C variable that is associated with `var` at state ω, is used for evaluating the expression; or
- not defined or derived within the C program entity and there exists a state ω of W_{prog} where ω maps to an evaluation of an expression where
  - the expression is not a function call expression and within the C program entity, and
  - the lvalue or rvalue of a C variable that is associated with `var` at state ω, is used for evaluating the expression.

4.4.2 Architecture Model of C Program Structure

The following theorem shows that the structural conditions of an architecture as presented in Definition 1 hold for the set of components $\mathscr{A}$ from Section 4.4.1.

Theorem 1. *Given a structure of a sequential non-recursive C program and a set of components $\mathscr{A}$ where the properties (I), (II-a), and (II-c)-(II-e) hold, it follows that:*

- *for any non-leaf node $\mathbb{E} = (X, \mathsf{B})$ of $\mathscr{A}$ where $\{(X_i, \mathsf{B}_i)\}_{i=1}^{N}$ are the children of $\mathbb{E}$, it holds that $X \subseteq \bigcup_{i=1}^{N} X_i$; and*
- *if there is a child $\mathbb{E}' = (X', \mathsf{B}')$ and a non-descendent $\mathbb{E}'' = (X'', \mathsf{B}'')$ of a component $\mathbb{E} = (X, \mathsf{B})$ in $\mathscr{A}$, such that $x \in X'$ and $x \in X''$, then it holds that $x \in X$.*

Proof. Consider a structure of a sequential non-recursive C program and a set of components $\mathscr{A}$ where the properties (I), (II-a), and (II-c)-(II-e) hold. In accordance with the property (I), the set $\mathscr{A}$ is organized as a rooted tree. Consider an arbitrary non-leaf component (X, B) in the tree, where (X, B) is mapped to a non-leaf C program entity in the C program structure. In accordance with the property (II-a), each port variable $x \in X \setminus \{x_{prog}\} \subseteq X_{F \cup V}$ models an identifier in the C program. Furthermore, in accordance with the properties (II-e), whether it holds that there exists a port variable $x \in X \setminus \{x_{prog}\}$ modeling a certain identifier or not, except for the special case with reference C variable identifiers and the identifier `main`, depends directly on the condition (a) and (b) of Theorem 1 and on which port variables that are included in the interfaces of the leaf components. Since a port variable modeling either a C variable reference identifiers or the identifier `main`, is also included in the interface of a component if it is included in the interface of one of its children, it follows that, in combination with the properties (II-c)-(II-d), the property (II-e) is sufficient to ensure that both the condition (a) and (b) of Theorem 1 hold. □

In accordance with Proposition 1 and Definition 1, it follows that $\mathscr{A}$ is an architecture if it holds that $\mathsf{B} = \widehat{proj}_X(\bigcap_{i=1}^{N} \mathsf{B}_i)$ for any non-leaf node $\mathbb{E} = (X, \mathsf{B})$ with children $\{(X_i, \mathsf{B}_i)\}_{i=1}^{N}$. Such an architecture is called an *architecture model of the C program structure.*

Definition 3 (Architecture Model of C Program Structure)**.** An *architecture model of a given C program structure* is an architecture of components where the properties (I)-(II) hold. □

The following theorem shows that the set of interfaces of the components in an architecture model is unique for the given C program structure.

Theorem 2. *Consider two architecture models $\mathscr{A}$ and $\mathscr{A}'$ of a given structure of a sequential non-recursive C program and also two components $(X, \mathsf{B}) \in \mathscr{A}$ and $(X', \mathsf{B}') \in \mathscr{A}'$ mapped from the same C program entity, it holds that the interfaces X and X' are identical, i.e. it holds that:*

i) the interfaces X and X' of the two components are partitioned into the same sets $X_1^F, \ldots, X_N^F$, X_{var}, and $\{x_{prog}\}$; and

ii) each port variable $x \in X \cup X'$ has the same label with respect to the components (X, B) and (X', B').

Proof. Consider two architecture models $\mathscr{A}$ and $\mathscr{A}'$ of a given structure of a sequential non-recursive C program and two components $(X, \mathsf{B}) \in \mathscr{A}$ and $(X', \mathsf{B}') \in \mathscr{A}'$ mapped from the same C program entity. In accordance with the property (II-a), each port variable $x \in X \setminus \{x_{prog}\} \subseteq X_{F \cup V}$ and $x \in X' \setminus \{x_{prog}\} \subseteq X_{F \cup V}$ model identifiers in the C program. Notably, the properties (II-c)-(II-e) that determine the partitioning of the interfaces X and X', as well as the mapping of their port variables to identifiers, are all of type 'if and only if'. That is, it either holds that a port variable is included from $X_{F \cup V}$ or it holds that it is not, and either is a subset of X and X' organized into a function interface or this is not the case. That is, it must hold that the condition (i) of Theorem 2 holds. Furthermore, in accordance with the property (II-b), it holds that the conditions for determining the labels of each port variable $x \in X \setminus \{x_{prog}\} \subseteq X_{F \cup V}$ and $x \in X' \setminus \{x_{prog}\} \subseteq X_{F \cup V}$ are mutually exclusive. Since the condition (i) ensures that it holds that $X = X'$, it follows that each port variable $x \in X \cup X'$ also has the same label with respect to (X, B) and (X', B'). That is, it follows that the condition (ii) of Theorem 2 holds. □

Consider the representation in Figure 2a of the architecture model of the structure of the C program `count10` in Figure 2b. Notably, in accordance with Section 4.3.1, the graphical syntax considers the organization of the

components and the port variables in their interfaces, the partitioning of the interfaces into function interfaces, and the labels of the port variables and their mapping. This means, in accordance with Theorem 2, that each architecture model of the structure of the C program `count10` in Figure 2b, would be represented in the exact same manner as shown in Figure 2a.

4.4.3 Discussion

As discussed in Section 4.3.4, there are cases where it is relevant to include other port variables than those explicitly expressed to be included in component interfaces by the properties (II-c)-(II-e) in Section 4.4.1. In such a case, in the context of a set of components $\mathscr{A}$ with the property (I), consider that a port variable is included in the interface of a component such that its interface is not such that the properties (II-c)-(II-e) hold. In order for $\mathscr{A}$ to also be in accordance with the listed conditions in Proposition 1, the port variable must also be in the interface of:

- a child of the component if the component is a non-leaf component; and
- the parent of the component if the port variable is also in the interface of a non-descendant of the component.

Notably, these conditions can easily be enforced by a tool such as the one that will be presented in Section 5 that now follows.

5 Feasibility Study on Automatic Extraction of Architecture Models

In order to determine the feasibility of automatically extracting architectural models from C program structures in accordance with Section 4.4, a prototype tool has been developed. The *LLVM Compiler Infrastructure* [18] was selected as the foundation for this prototype tool. The LLVM framework consists of a front-end tool, a transformation tool, and a back-end tool. The front-end tool, which in the case of the C programming language is called *Clang*, translates the source code files into an intermediate representation (IR). The transformation tool, named *Opt*, allows for IR-to-IR transformations to be performed, most commonly to perform program optimization.

The back-end tool, called *llvm*, translates the IR into machine level code for a given architecture.

The design outline of the developed prototype tool is presented in Figure 7. The tool utilizes artifacts produced by Clang as part of the normal compilation process, and adds a custom transformation pass to Opt, in order to extract architectural concepts into a raw data file. This raw data file can then be consumed by filters to produce architectural representations. Figure 7 shows two such filters, namely a filter producing a graphical call-graph and a filter producing an XML-based representation that describes interfaces of components in accordance with Definition 2.

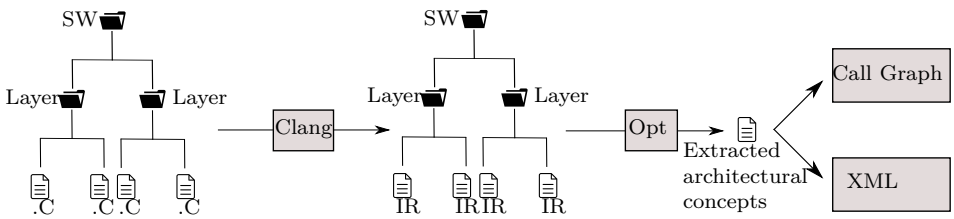


Figure 7. Automatic architecture recovery tool design.

5.1 Source Code to IR

The prototype tool takes an input in the form of a folder structure where the lowest-level folders contain .c-files. The Clang tool translates each individual .c-file into a corresponding IR translation unit. This step is performed while preserving the folder structure of the original source code tree. The folder structure containing the IR translation units, correspond to a C program structure as described in Section 3.3 where each folder corresponds to a non-leaf node of the C program structure. No compiler optimization is performed as part of this step.

5.2 IR to Extracted Architectural Concepts

A custom extraction pass has been created for the Opt tool in order to extract architectural concepts from the C program structure corresponding to the folder structure containing the IR translation units. The pass is executed once per IR translation unit, with a single post-processing pass

that is executed once all translation units have been processed. For each IR translation unit, the following data is gathered:

i) a list of C variable identifiers that are defined within the translation unit, but outside of definitions of C function identifiers, created by iterating over the IR list of global variables produced by opt;

ii) a list of non-zero values that pointers are initialized to;

iii) a list of C function identifiers defined within the translation unit is created, containing the name of each C function identifier, its type and formal parameters; and

iv) a list that associates each C function identifier in the list (iii) with two lists:

- a list containing each C variable identifier that is
  - read from or written to in the definition of the C function identifier, and
  - defined outside of any definitions of C function identifiers; and
- a list containing each C function identifier that is called in the definition of the C function identifier.

The list (i) contains defined and derived C variable identifiers and the list (ii) contains C variable reference identifiers as described in Section 3.2. The list (iii) contains declarations of C function identifiers defined within the translation unit.

The list (iv) captures external dependencies that the execution of code within definitions of C function identifiers defined within the translation unit, has to C variable and C function identifiers. The dependencies to C variable identifiers in the list (iv) are established by iterating over the instructions that together make up definitions of C function identifiers, and identifying whether each instruction is a load instruction or a store instruction. Note that the identified dependencies are exclusive to *global* C variable identifiers, i.e. C variable identifiers that are *not* defined within definitions of C functions. *Memory Dependence Analysis*, an analysis pass that is part of the LLVM framework, is used to identify C variable identifiers that are

indirectly read from or written to, i.e. through the use of pointers. However, these indirect reads/writes can only be identified if the pointers are constant. These limitations are further discussed in Section 5.3

Once each translation unit has been processed individually, the post-processing pass is performed where a tree data structure is first created by considering the folder structure and each list (iii) that specifies where different C function identifiers are defined. Each node of the tree data structure corresponds to a component. Subsequently, for each node in the tree data structure, by considering the lists (iv) for the different translation units and the properties in Sections 4.3.2 and 4.3.3, a subset of the union of the lists (i)-(iii) for the translation units, is associated as port variables of the node. In accordance with Section 4.1, labels of the port variables with respect to each node, are established by considering the lists (i) for the different translation units, and distinguishing whether its members have scopes that extends outside of the translation unit or not.

The tree data structure of components, produced by the post-processing pass is serialized and written to a temporary raw data file. As previously explained, this temporary raw data file can be used to generate different architectural representations.

5.3 Discussion and Future Challenges

While many of the architectural concepts presented within this paper can be successfully extracted by the prototype tool, a few challenges were identified during its development. Most of these challenges are possible to address by complementing current analyzes with e.g. analyzes on an AST of a C program. However, other challenges, e.g. determining target addresses of non-constant pointers, are essentially unsolvable in general. The following will discuss the most prominent challenges in more detail.

Local variables As mentioned in Section 5.2, the prototype tool can only identify dependencies regarding C variable identifiers that are defined outside of definitions of C function identifiers, i.e. *global* C variable identifiers, and not those that are local. This is due to local C variable identifiers in the LLVM IR being represented in Static Single Assignment (SSA) form [51], which makes it difficult to properly extract function-local names of identifiers since they are renamed whenever they are subject to a new value

assignment. This information would need to be extracted from an AST of the C program, and in fact, such an AST can be generated from Clang.

Indirect Reads/Writes Using Non-constant pointers Section 5.2 mentioned that the prototype tool is currently limited in only being able to identify indirect reads/writes through the use of pointers, if the pointers are constant. Notably, determining how the target addresses of non-constant pointers, i.e. pointers that can be assigned other targets during program execution, change during execution is a general issue [52]. Moreover, if non-constant pointers are used in combination with complex control flows, it is essentially infeasible to automatically extract an architecture model of a C program in accordance with Definition 3. That is, to fully enable automatic extraction, the only solution is to restrict the use of such complex communication patterns; examples of such restrictions can be found in the standard MISRA-C [53] or in ISO 26262 [12] that e.g. limits the use of pointers and forbids unconditional jumps, as well as hidden data and control flows.

6 Conclusion

Architecture models are essential in order to manage the complexity of C programs in general, and to support expressing and organizing specifications for C program structures, in particular. However, in order to create architecture models that accurately describe structures of C programs, support is needed in the form of a formal mapping from C program structures to architecture models. Considering that such support is lacking in current modeling approaches/languages [5–10, 19, 21], the present paper has established a formal architecture model of C program structures.

More specifically, for a given structure of a sequential non-recursive C program, a set of mapping properties were presented. These properties were shown to uniquely map to a set of interfaces of components organized in accordance with the structure of the C program. An architecture of components with these properties was defined as an *architecture model of the C program structure*. The interface of each component in the architecture model consists of port variables modeling C function and C variable identifiers. Hence, the architecture model of the C program structure provide a foundation both for organizing specifications in accordance with the structure of the C program and for expressing specifications in terms of identifiers

in the C program. Additionally, for practical application, a graphical syntax was presented for representing architecture models of C program structures.

The mapping properties enable to establish an architecture model of complex legacy code to which no prior architecture model exists. Furthermore, supposing that an architecture model does already exist, it might have become inconsistent over time [54]; thus, the mapping properties can be used to verify the consistency of the architecture model. In the case where an architecture of components is established prior to implementing code, a mapping from identifiers to port variables can be specified explicitly. In such a case, the architecture serves as a design specification, rather than as a description of an existing C program structure.

In order to determine the feasibility of automatically extracting architecture models from a C program structure, a prototype tool was presented. Conclusion from this was that, except for already well-known issues regarding indirect reads/writes through pointers, the architectural concepts can successfully be automatically extracted.

Acknowledgements

This work is a result from a collaborative effort with automotive manufacturer *Scania CV AB*, and has received funding under grant agreement no.2011 − 04446 from Swedish governmental agency for innovation *VINNOVA*.

A special thanks to Maxim Olifer who implemented parts of the architecture recovery prototype tool.

References

[1] M. Broy, I. H. Kruger, A. Pretschner, C. Salzmann, Engineering Automotive Software, Proceedings of the IEEE 95 (2) (2007) 356–373, ISSN 0018-9219, doi:10.1109/JPROC.2006.888386.

[2] B. W. Kernighan, The C Programming Language, Prentice Hall Professional Technical Reference, 2nd edn., ISBN 0131103709, 1988.

[3] ISO/IEC/IEEE 42010, System and software Eng. - Architecture description, 2011.

[4] T. Mens, J. Magee, B. Rumpe, Evolving Software Architecture Descriptions of Critical Systems, Computer 43 (5) (2010) 42–48, ISSN 0018-9162, doi:10.1109/MC.2010.136.

[5] S. Friedenthal, A. Moore, R. Steiner, A Practical Guide to SysML: Systems Modeling Language, Morgan Kaufmann Inc., San Francisco, CA, USA, ISBN 0123743796, 9780080558363, 9780123743794, 2008.

[6] J. Rumbaugh, I. Jacobson, G. Booch, Unified Modeling Language Reference Manual, The (2nd Edition), Pearson Higher Education, ISBN 0321245628, 2004.

[7] P. H. Feiler, D. P. Gluch, Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, Addison-Wesley Professional, 1st edn., ISBN 0321888944, 9780321888945, 2012.

[8] S. Chaki, E. Clarke, A. Groce, S. Jha, H. Veith, Modular Verification of Software Components in C, in: Proceedings of the 25th International Conference on Software Engineering, ICSE '03, IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-1877-X, 385–395, URL http://dl.acm.org/citation.cfm?id=776816.776863, 2003.

[9] T. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Software Verification with BLAST, in: T. Ball, S. Rajamani (Eds.), Model Checking Software, vol. 2648 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, ISBN 978-3-540-40117-9, 235–239, doi:10.1007/3-540-44829-2_17, URL http://dx.doi.org/10.1007/3-540-44829-2_17, 2003.

[10] D. Greenaway, Automated proof-producing abstraction of C code, Ph.D. thesis, University of New South Wales, 2014.

[11] S. Cass, Top 10 Programming Languages, URL http://spectrum.ieee.org/computing/software/top-10-programming-languages, 2014.

[12] ISO 26262, Road vehicles-Functional safety, 2011.

[13] W. de Roever, H. Langmaack, A. Pnueli, Compositionality: The Significant Difference, Springer, ISBN 9783540654933, 1998.

[14] J. Hooman, W. P. de Roever, The Quest Goes on: A Survey of Proofsystems for Partial Correctness of CSP, in: Current Trends in Concurrency, Overviews and Tutorials, Springer Berlin Heidelberg, 343–395, doi:10.1007/BFb0027044, URL http://dx.doi.org/10.1007/BFb0027044, 1986.

[15] M. W. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. P. Heimdahl, S. Rayadurgam, Your What Is My How: Iteration and Hierarchy in System Design, IEEE Software 30 (2) (2013) 54–60, ISSN 0740-7459, doi:http://doi.ieeecomputersociety.org/10.1109/MS.2012.173.

[16] G. C. Murphy, D. Notkin, K. Sullivan, Software reflexion models: bridging the gap between source and high-level models, SIGSOFT Softw. Eng. Notes 20 (4) (1995) 18–28, ISSN 0163-5948, doi:10.1145/222132.222136, URL http://doi.acm.org/10.1145/222132.222136.

[17] G. Rasool, N. Asif, Software architecture recovery, International Journal of Computer, Information, and Systems Science, and Engineering 1 (3).

[18] C. Lattner, V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, in: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, 2004.

[19] S. Soleimanifard, D. Gurov, Algorithmic verification of procedural programs in the presence of code variability, Science of Computer Programming (2015) –ISSN 0167-6423, doi:http://dx.doi.org/10.1016/j.scico.2015.08.010, URL http://www.sciencedirect.com/science/article/pii/S0167642315002592.

[20] K. Laster, O. Grumberg, Modular model checking of software, in: B. Steffen (Ed.), Tools and Algorithms for the Construction and Analysis of Systems, vol. 1384 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, ISBN 978-3-540-64356-2, 20–35, doi:10.1007/BFb0054162, URL http://dx.doi.org/10.1007/BFb0054162, 1998.

[21] R. Alur, R. Grosu, Modular Refinement of Hierarchic Reactive Machines, ACM Trans. Program. Lang. Syst. 26 (2) (2004) 339–369, ISSN 0164-0925, doi:10.1145/973097.973101.

[22] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, C. Sofronis, Multiple Viewpoint Contract-Based Specification and Design, in: F. de Boer, M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Formal Methods for Components and Objects, vol. 5382 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, ISBN 978-3-540-92187-5, 200–225, doi:10.1007/978-3-540-92188-2_9, URL http://dx.doi.org/10.1007/978-3-540-92188-2_9, 2008.

[23] A. L. Sangiovanni-Vincentelli, W. Damm, R. Passerone, Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems, European Journal of Control 18 (3) (2012) 217–238.

[24] J. Westman, M. Nyberg, Environment-Centric Contracts for Design of Cyber-Physical Systems, in: J. Dingel, W. Schulte, I. Ramos, S. Abrahao, E. Insfran (Eds.), Model-Driven Engineering Languages and Systems, vol. 8767 of *Lecture Notes in Computer Science*, Springer International Publishing, ISBN 978-3-319-11652-5, 218–234, doi:10.1007/978-3-319-11653-2_14, 2014.

[25] J. Westman, M. Nyberg, Contracts for Specifying and Structuring Requirements on Cyber-Physical Systems, in: D. B. Rawat, J. Rodriques, I. Stojmenovic (Eds.), Cyber Physical Systems: From Theory to Practice, Taylor & Francis, ISBN 9781482263329, 2015.

[26] T. Henzinger, J. Sifakis, The Discipline of Embedded Systems Design, Computer 40 (10) (2007) 32–40, ISSN 0018-9162, doi:10.1109/MC.2007.364.

[27] E. Lee, Cyber Physical Systems: Design Challenges, in: Object Oriented Real-Time Distributed Computing (ISORC), 11th IEEE Int. Symp. on, 363–369, doi:10.1109/ISORC.2008.25, 2008.

[28] D. B. Rawat, J. J. Rodrigues, I. Stojmenovic, Cyber-Physical Systems: From Theory to Practice, CRC Press, 2015.

[29] A. Baumgart, P. Reinkemeier, A. Rettberg, I. Stierand, E. Thaden, R. Weber, A Model-Based Design Methodology with Contracts to Enhance the Development Process of Safety-Critical Systems, in: Software Technologies for Embedded and Ubiquitous Systems, vol. 6399 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, ISBN 978-

3-642-16255-8, 59–70, doi:10.1007/978-3-642-16256-5_8, URL http://dx.doi.org/10.1007/978-3-642-16256-5_8, 2011.

[30] W. Damm, B. Josko, T. Peinkamp, Contract Based ISO CD 26262 Safety Analysis, in: Safety-Critical Systems, 2009, SAE, doi:10.4271/2009-01-0754, 2009.

[31] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, I. Stierand, Using contract-based component specifications for virtual integration testing and architecture design, in: Design, Automation Test in Europe Conference Exhibition (DATE), 2011, ISSN 1530-1591, 1–6, doi:10.1109/DATE.2011.5763167, 2011.

[32] J. Westman, M. Nyberg, A Reference Example on the Specification of Safety Requirements using ISO 26262, in: M. ROY (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, France, NA, URL http://hal.archives-ouvertes.fr/hal-00848610, 2013.

[33] J. Westman, M. Nyberg, M. Törngren, 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, Springer-Verlag New York, Inc., New York, NY, USA, ISBN 978-3-642-40792-5, 166–177, doi:10.1007/978-3-642-40793-2_16, URL http://dx.doi.org/10.1007/978-3-642-40793-2_16, 2013.

[34] J. Westman, M. Nyberg, Extending Contract theory with Safety Integrity Levels, in: HASE, 2015 IEEE 16th Int. Symposium on, 85–92, doi:10.1109/HASE.2015.21, 2015.

[35] C. Chilton, B. Jonsson, M. Kwiatkowska, An algebraic theory of interface automata, Theoretical Computer Science 549 (2014) 146–174.

[36] D. L. Dill, Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits, in: Proceedings of the fifth MIT conference on Advanced research in VLSI, MIT Press, Cambridge, MA, USA, ISBN 0-262-01100-X, 51–65, URL http://dl.acm.org/citation.cfm?id=88056.88061, 1988.

[37] E. S. Wolf, Hierarchical Models of Synchronous Circuits for Formal Verification and Substitution, Ph.D. thesis, Stanford University, Stanford, CA, USA, uMI Order No. GAX96-12052, 1996.

[38] S. D. Brookes, C. A. R. Hoare, A. W. Roscoe, A Theory of Communicating Sequential Processes, J. ACM 31 (3) (1984) 560–599, ISSN 0004-5411, doi:10.1145/828.833, URL http://doi.acm.org/10.1145/828.833.

[39] R. Negulescu, Process Spaces, in: Proceedings of the 11th Int. Conf. on Concurrency Theory, CONCUR '00, Springer-Verlag, London, UK, UK, ISBN 3-540-67897-2, 199–213, URL http://dl.acm.org/citation.cfm?id=646735.701627, 2000.

[40] E. F. Codd, A Relational Model of Data for Large Shared Data Banks, Commun. ACM 13 (6) (1970) 377–387, ISSN 0001-0782, doi:10.1145/362384.362685, URL http://doi.acm.org/10.1145/362384.362685.

[41] L. Lamport, A Simple Approach to Specifying Concurrent Systems, Commun. ACM 32 (1) (1989) 32–45, ISSN 0001-0782, doi:10.1145/63238.63240, URL http://doi.acm.org/10.1145/63238.63240.

[42] M. Abadi, L. Lamport, The Existence of Refinement Mappings, Theor. Comput. Sci. 82 (2) (1991) 253–284, ISSN 0304-3975, doi:10.1016/0304-3975(91)90224-P, URL http://dx.doi.org/10.1016/0304-3975(91)90224-P.

[43] J. B. Dabney, T. L. Harman, Mastering simulink, Pearson/Prentice Hall, 2004.

[44] B. Josko, Q. Ma, A. Metzner, Designing Embedded Systems using Heterogeneous Rich Components, in: Proceedings of the INCOSE International Symposium, 2008.

[45] R. Diestel, Graph Theory, 4th Edition, vol. 173 of *Graduate texts in mathematics*, Springer, ISBN 978-3-642-14278-9, 2012.

[46] M. Abadi, L. Lamport, Composing specifications, ACM Trans. Program. Lang. Syst. 15 (1) (1993) 73–132, ISSN 0164-0925, doi:10.1145/151646.151649, URL http://doi.acm.org/10.1145/151646.151649.

[47] R. Alur, T. Henzinger, Reactive Modules, Formal Methods in System Design 15 (1) (1999) 7–48, ISSN 0925-9856, doi:10.1023/A:1008739929481, URL http://dx.doi.org/10.1023/A%3A1008739929481.

[48] AUTOSAR, AUTomotive Open System ARchitecture, URL http://www.autosar.org/, 2015.

[49] ISO 7498-1, Information technology - OSI - Basic Reference Model, 1994.

[50] M. Nyberg, Failure propagation modeling for safety analysis using causal Bayesian networks, in: Control and Fault-Tolerant Systems (SysTol), 2013 Conference on, 91–97, doi:10.1109/SysTol.2013.6693936, 2013.

[51] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, ACM Trans. Program. Lang. Syst. 13 (4) (1991) 451–490, ISSN 0164-0925, doi:10.1145/115372.115320, URL http://doi.acm.org/10.1145/115372.115320.

[52] M. Hind, Pointer Analysis: Haven'T We Solved This Problem Yet?, in: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01, ACM, New York, NY, USA, ISBN 1-58113-413-4, 54–61, doi:10.1145/379605.379665, URL http://doi.acm.org/10.1145/379605.379665, 2001.

[53] MISRA C:2012 - , Guidelines for the Use of the C Language in Critical Systems, 2013.

[54] J. van Gurp, J. Bosch, Design erosion: problems and causes, Journal of Systems and Software 61 (2) (2002) 105 – 119, ISSN 0164-1212, doi:http://dx.doi.org/10.1016/S0164-1212(01)00152-2, URL http://www.sciencedirect.com/science/article/pii/S0164121201001522.

Paper D

Providing Tool Support for Specifying Safety-Critical Systems by Enforcing Syntactic Contract Conditions

Jonas Westman • Mattias Nyberg

Extended version of paper submitted to *Requirements Engineering*.

Abstract

Functional safety (FuSa) standards such as IEC 61508 and ISO 26262 advocate a particularly stringent *Requirements engineering* (RE) where safety requirements must be *structured* in an hierarchical manner and *specified* in accordance with the system *architecture*. In contrast to the stringent RE in FuSa standards, according to previous studies, RE in industry is in general of poor quality. *Contracts theory* has been previously shown to be suitable for supporting such a stringent RE 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 RE practice and the stringent RE in FuSa standards, it is shown how support in a specification tool can be provided even when requirements, and also architectures, are not formalized. This is achieved by enforcing *syntactic*, yet formal, conditions in contracts theory. Furthermore, a validating industrial case study is presented where the proposed support is shown to be fully applicable in an industrial setting.

1 Introduction

Requirements engineering (RE) [15, 39] 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 [41]. In fact, the general FuSa standard IEC 61508 [40] 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) [1, 25, 40, 41] that specifies the required *reliability* [68] of a system or component, in order to achieve a tolerable level of risk. FuSa is a key property of *heterogeneous systems* [36, 53, 69], i.e. systems such as airplanes and automotive vehicles that are composed of parts from multiple domains, e.g. software (SW), hardware (HW), mechanical, electrical, etc.

In IEC 61508 and its derivative FuSa standards such as ISO 26262 [41] for the automotive domain, safety requirements must be structured in an hierarchical manner in accordance with the *system architecture* [42]: at each level, safety requirements must be *allocated* to *architecture elements* and *trace links* [19] 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"* [41]. This is a property that also must be verified, 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 [4] and are typically incomplete [28], and this is also true for safety requirements [28, 54]. 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 [74], which is shared in the present paper, is that RE is a 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 describes how such tool support can be provided when authoring specifications for heterogeneous systems in an industrial setting. This is achieved by applying the work in [8, 72, 77, 79] that present a formal *contracts* [57] theory for modeling and specifying heterogeneous systems.

In particular, this contracts theory contains a concept called a *contract structure* that captures an 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 the contracts theory [8, 72, 77, 79], these tasks are given *formal*

semantics, i.e. interpretations in 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 enforces these conditions.

Notably, there already exist approaches such as [17, 18] and [20] where formal contract conditions are enforced 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) [64]. Despite the fact that formal representations have several advantages over non-formal ones, formal languages are difficult to use by non-experts [26] and in industrial practice, *"overcoming the burden of formalization is a major challenge"* [13]. Therefore, instead of focusing on enforcing all of the contract conditions in [8, 72, 77, 79], the present paper instead identifies necessary conditions of the formal interpretations of tasks (I)-(IV) where these conditions can be enforced even when requirements and architectures are not represented formally; in the following, such necessary conditions will be called *syntactic contract conditions* since they can be enforced without explicitly considering the formal semantics of tasks (I)-(IV).

As the main contribution, the present paper shows how a specification tool will provide *feedback*- and *guidance*-driven support for tasks (I)-(IV) by enforcing their syntactic contract conditions. It is shown that such support for tasks (I), (II), and (IV) can be provided regardless of representation format of requirements and when architectures are represented formally or *semi-formally*, as an hierarchy of interfaces consisting of port variables. For task (III), feedback and guidance can be provided when architectures are represented formally or semi-formally and when requirements are represented formally or *semi-formally*, i.e. as free text with distinguishable port variables.

As a second contribution and validation of the main contribution, it is described how the proposed feedback- and guidance-driven support can be implemented in an industrial setting. This is achieved by describing the design and implementation solutions used for realizing this support in a validating industrial case study performed at Scania – a global heavy trucks manufacturer located in Sweden. A key necessary concept for realizing this support in practice is shown to be Linked Data [12], which enables *formal referencing* in between specifications and to architecture data; formal referencing is necessary since the syntactic contract conditions are indeed formal, and thus, the conditions cannot be properly evaluated if it is unclear what data a reference points to in a specification. However, not only that, but

Linked Data is also shown to increase quality of specification in general by ensuring consistency of data presented in specifications.

Notably, there are other works such as [26, 32, 48] 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 [26, 32, 48] 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 [26, 32, 48] improves readability of requirements, the approach in the present paper enforces their correctness and thus, the approaches complement each other.

In contrast to [26, 32, 48], but in accordance with the present paper, the works [5, 33, 34] describe formally founded support for RE. While being similar in their fundamental approaches, the present paper and [34] have different focuses and complement each other; the present paper focuses on tool support for tasks (I)-(IV) while [34] focuses on transformation between requirements specified in NL, the formal representation format Object Constraint Language (OCL) [75], and a semi-formal representation format in between these.

More similar to the present paper, the works [5, 33] both focus on establishing trace links between requirements and design/architecture. However, while the approach in the present paper is applicable for any type of development methodology, e.g. waterfall, v-model, model driven development, etc., the approach in [5] is only applicable when using model transformations for driving development. The work in [33] present support for validating and generating trace links; however, in contrast to the present paper, the support in [33] requires formal representations of architectures and requirements. The support in the present paper caters to needs in current state of industrial practice where requirements and architectures are typically not represented formally and where development methodologies vary, sometimes even within the same company.

The rest of this paper is organized such that it continuously moves from theory, through design, to implementation. Section 2 introduces relevant parts of the contracts theory in [8, 72, 77, 79]. Section 3 then first identifies syntactic contract conditions of the formal interpretation of tasks (I)-(IV) in this contracts theory. Second, and as the main contribution, it is shown how a conceptual specification tool will provide feedback- and guidance-driven

support for tasks (I)-(IV) by enforcing these syntactic contract conditions. Section 4 then describes how the main contribution can be realized in an industrial setting by describing the design and implementation solutions used in the validating industrial case study. Section 5 summarizes the present paper and draws conclusions.

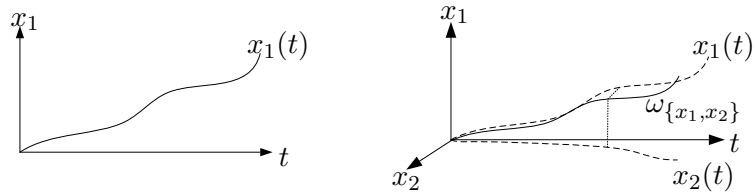
2 Contracts Theory for Safety-Critical Heterogeneous Systems

The notion of *contracts* was first introduced in [57] for formal specification of SW. However, the principles behind contracts can be traced back to early ideas on *compositional* [38, 70] proof-methods [37, 45, 58]. In [8, 72], the use of contracts is extended from formal specification of SW to serving as a central systems engineering philosophy to support the design of heterogeneous systems. The work in [77] incorporates the work in [8, 72] and presents a contracts theory that introduces new concepts such as *architecture*. This contracts theory is extended in [79] to a safety-critical context with the notion of SILs.

As previously mentioned in Section 1, Section 3 will show how tool support can be provided for tasks (I)-(IV) by enforcing conditions from the contract theory in [8, 72, 77, 79]. This section not only summarizes these conditions and related concepts from this contracts theory, but also describes them thoroughly. Describing them thoroughly is considered to be necessary in order to fully understand the type of tool support that can be provided when enforcing these conditions without first having to read the papers [8, 72, 77, 79]. Note that the proposed use-cases/motivations in these papers differ from the overall aim in providing tool support for tasks (I)-(IV) in the present paper. Thus, these concepts and conditions will be presented from a different perspective in the present paper and are also slightly tailored to better fit this overall aim.

2.1 Assertions and Runs

The theory [8, 72, 77, 79] relies on a general formalism called *assertions*, i.e. sets of value sequences called *runs*, which are used for expressing requirements and behaviors. This section describes the concept of assertions and runs in more detail.



(a) (b)

Figure 1. In (a), a trajectory of values of x_1 is shown. In (b), a run $\omega_{\{x_1,x_2\},T}$ is shown, consisting of two pairs containing the trajectory shown in (a) and another trajectory of values of x_2.

2.1.1 Runs

Let $X = \{x_1, \ldots, x_N\}$ be a non-empty set of variables, each with its own domain. Consider a pair (x_i, ξ_i) consisting of a variable x_i and a trajectory $\xi_i = \{(t, x_i(t))\}_{t \in T}$ of values $x_i(t)$ of x_i over a time-window $\emptyset \neq T \subseteq \mathbb{R}_{\geq 0}$. For example, Figure 1a shows a trajectory of values of the variable x_1. A set of such pairs $\{(x_1, \xi_1), \ldots, (x_N, \xi_N)\}$ with trajectories over the same time-window T, is called a *run* for X over T, denoted either $\omega_{X,T}$ or simply ω.

For example, a run can be a *trace* [14, 16, 24, 81] or an *execution* as presented in [60]. As a more illustrative example, a run $\omega_{\{x_1,x_2,\},T}$ consisting of two pairs is shown in Figure 1b as a solid line where the trajectories of the pairs are also shown as two dashed lines. The trajectory of values of x_1 is the trajectory shown in Figure 1a and the other trajectory consists of values of variable x_2.

2.1.2 Assertions

In the following, assume a universal set of variables Ξ and a domain for each variable $x \in \Xi$. Let Ω denote the set of all possible runs for Ξ over each time-window $\emptyset \neq T \subseteq \mathbb{R}_{\geq 0}$. An *assertion* W is, a possibly empty, subset of Ω, i.e. $\mathsf{W} \subseteq \Omega$.

An assertion can be can be syntactically represented by constraints, e.g. by equations, inequalities, or logical formulas. For example, an assertion W' represented by the equation $u = v$, is the set of all runs in Ω where $u = v$

holds for each time point.

As a second example of how an assertion can be syntactically represented, consider that W'' is represented by the first order differential equation

$$\frac{\mathrm{d}y}{\mathrm{d}t} = x(t), \text{ where } x(t) = t \text{ and } t \in \mathbb{R}_{\geq 0} \ .$$

The assertion W'' is the set of all possible runs for Ξ over $\mathbb{R}_{\geq 0}$ where the runs are the solutions to the differential equation.

As a third example of how an assertion can be syntactically represented, consider that W''' is an assertion represented by the logic formula $a(t) = 0 \vee b(t) = 0$ where $t \in \{0, 1, \ldots, 10\}$ and where both variables a and b take values from $\{0, 1\}$. This means that the assertion W''' is the set of all possible runs for Ξ over the discrete time-window $\{0, 1, \ldots, 10\}$ where, for each time-point t, at least one of a and b has the value 0.

2.1.3 Variables Constrained by Assertions

As previously presented, assertions are sets of runs for the universal set of variables Ξ. However, as can be seen in previous examples, assertions can be represented by constraints specified over a subset of Ξ. For example, the equation $u = v$, representing the previously mentioned assertion W', is specified simply over the set of variables $\{u, v\}$. In fact, the set $\{u, v\}$ is actually the set of variables necessary for representing W' syntactically; considering an arbitrary assertion W, such a set of variables is called *the set of variables constrained by* W.

Prior to presenting a formal definition of such a set of variables, the concept of *projections* [9, 73, 77, 79, 80] needs to be introduced. The *projection* [9, 73, 77, 79, 80] of W onto X, written $proj_X(\mathsf{W})$, is the set obtained when each pair that does not contain a variable $x \in X$ is removed from each run $\omega_{\Xi,T}$ in W, i.e.

$$proj_X(\mathsf{W}) = \{\{(x, \xi) \mid (x, \xi) \in \omega_{\Xi,T} \text{ and } x \in X\} \mid \omega_{\Xi,T} \in \mathsf{W}\} \ .$$

The *extended projection* [77] of W onto X is denoted $\widehat{proj}_X(\mathsf{W})$ and is the *inverse projection* [9, 80] of $proj_X(\mathsf{W})$ onto Ξ. That is,

$$\widehat{proj}_X(\mathsf{W}) = \{\omega \in \Omega \mid proj_X(\{\omega\}) \subseteq proj_X(\mathsf{W})\} \ .$$

Conceptually, extended projection corresponds to the notion of *port elimination* [8] or *variable hiding* [2, 49] through existential quantification.

Definition 1 (Variables Constrained by Assertion)**.** *A variable x is constrained by an assertion* W if

$$\widehat{proj}_{\Xi\setminus\{x\}}(\mathsf{W}) \neq \mathsf{W} \ .$$

Let X_{W} denote *the set of variables constrained by* W. □

For example, $X_{\mathsf{W}'} = \{u, v\}$ where $u = v$ represents W'.

2.2 Elements

This section introduces *elements* [77, 79], which can represent any part of a heterogeneous system in general, but can also serve as a *connection*, e.g. as described in Modelica [30, 31], or as a functional or logical design entity in general, e.g. as a Systems Modeling Language (SysML) block [29]. The concept of an element generalizes Heterogeneous Rich Components (HRCs) [11, 22, 46] as used in [8, 72].

Definition 2 (Element)**.** 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 Figure 2a as a rectangle filled with gray and 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 the equation $h = \frac{v_{branch} - v_{gnd}}{v_{ref} - v_{gnd}}$.

2.3 Architecture

A set of elements can be organized into an *architecture* [77, 79], which describes an hierarchical nesting of elements. This hierarchical nesting can

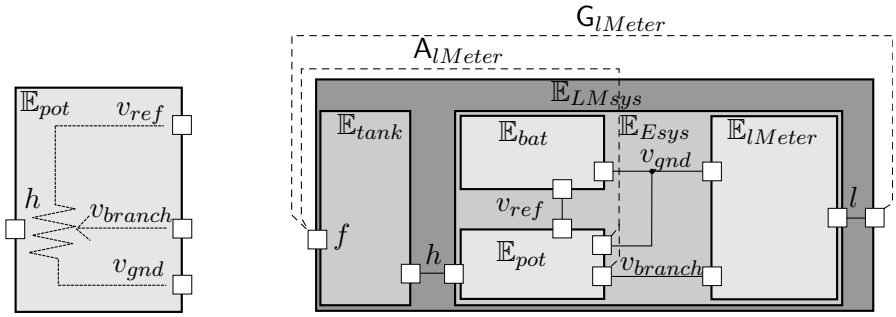


Figure 2. In a), an element $\mathbb{E}_{pot} = (X_{pot}, \mathsf{B}_{pot})$, representing a potentiometer, is shown. In b), an architecture, representing a "Level Meter system" and its parts, and a contract $\mathcal{C}_{lMeter} = (\{\mathsf{A}_{lMeter}\}, \mathsf{G}_{lMeter}, X_{lMeter})$, are shown.

be viewed as a rooted tree; thus, in the following, terminology from graph theory [23] will be borrowed to describe positions of elements in an architecture, relative to each other. The underlying principle is to combine individual behaviors using *intersection* [3, 8, 72] where the *sharing of port variables* [6, 8, 72] 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 Figure 2b. 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 Figure 2a, a battery $\mathbb{E}_{bat}$, and a level meter $\mathbb{E}_{lMeter}$. 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.

In the LM-system, the slider h 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 the 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 extended projection of the intersection of the behaviors of its children onto

X. 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 represented by the equation $l = h$.

Definition 3 (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) $\mathsf{B} = \widehat{proj}_X(\bigcap_{i=1}^{N} \mathsf{B}_i)$; and

b) if there is a child $\mathbb{E}' = (X', \mathsf{B}')$ and a non-descendent $\mathbb{E}'' = (X'', \mathsf{B}'')$ 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 Figure 2b.

As expressed in part (a)-(i) of Definition 3, the interface of an element $\mathbb{E}$ is a subset of the union of the interfaces of its children. Part (a)-(ii) expresses that the individual behaviors of the children of $\mathbb{E}$ are combined and abstracted with intersection and extended projection onto the interface of $\mathbb{E}$. Part (b) expresses that if a port variable x is in the interface of both a child of $\mathbb{E}$ and an element in the environment of $\mathbb{E}$, then x must also be in the interface of $\mathbb{E}$.

2.4 Contracts

A *contract* [8, 72, 77, 79] $(\{\mathsf{A}_i\}_{i=1}^{N}, \mathsf{G}, X)$ specifies 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 4 (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 $\mathbb{E} = (X', \mathsf{B})$ *satisfies* [8, 9, 11, 72, 77, 79] 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 Figure 2b. The guarantee G_{lMeter}, represented by the 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 the equation $f = \frac{v_{branch} - v_{gnd}}{5V}$. In accordance with conditions (1) and (2), the contract $\mathcal{C}_{lMeter}$ is satisfied by the element $\mathbb{E}_{lMeter}$.

2.5 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:

$$\text{if each } \mathbb{E}_i \text{ satisfies } \mathcal{C}_i \text{ then } \mathbb{E} \text{ satisfies } \mathcal{C}. \tag{3}$$

If an architecture only consists of two hierarchical levels, then property (3) corresponds to *dominance/refinement* of contracts as described in [7–10, 66, 72], the basic idea of *compositionality* [38, 70], and in particular, the notion of *completeness* in ISO 26262 [41]. 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 [77].

This section presents concepts originating from [79] that introduces a graph, called a *contract structure* that organizes contracts with the intent of having the property as expressed in property (3). Prior to presenting this formal definition of contract structure in Section 2.5.2, an underlying concept of using a contract to express a relation between requirements is described in Section 2.5.1. Sufficient conditions on a contract structure to achieve property (3) is presented in Section 2.5.3.

2.5.1 Contracts as Requirement Relations

Regard the contract $\mathcal{C}_{lMeter}$ for the element $\mathbb{E}_{lMeter}$, as shown in Figure 2b. In accordance with Section 2.4, 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 [80] where guarantees are used to express safety requirements on elements.

Consider a scenario where the environment that the element $\mathbb{E}_{Lmeter}$ is to be deployed in, is unknown, e.g. when developing $\mathbb{E}_{Lmeter}$ *"out-of-context"* [41]. The assumption A_{Lmeter} of the contract shown in Figure 2b 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 Figure 2b, *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 [76, 80] 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 Section 2.5.2, 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* $\mathbb{E}$. How to capture the cases where the use of explicit assumptions are needed, e.g. when it is necessary to expresses that an assumption is to be fulfilled by two or more guarantees, is explained in [77].

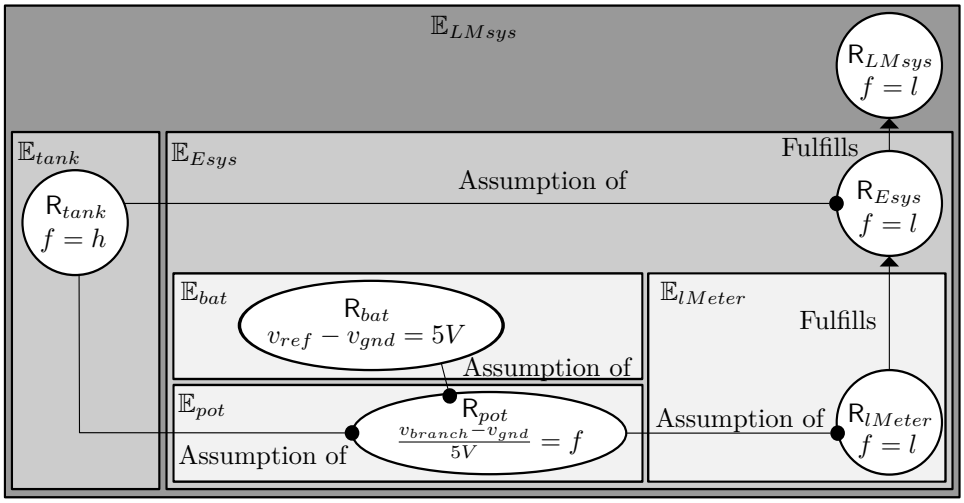


Figure 3. A contract structure for the architecture shown in Figure 2b.

2.5.2 Contract Structure for Architecture

Consider a set of contracts $\{\mathcal{C}_{LMsys}, \mathcal{C}_{Esys}, \mathcal{C}_{pot}, \mathcal{C}_{bat}, \mathcal{C}_{lMeter}\}$ where the guarantee of each contract $\mathcal{C}_i$ is a requirement R_i allocated to an element $\mathbb{E}_i$ in the architecture shown in Figure 2b. Consider these guarantees being structured as nodes in an edge-labeled directed graph as an overlay onto the hierarchical structure of the elements shown in Figure 3.

A guarantee R_i in a contract $\mathcal{C}_i$ is an assumption of another contract $\mathcal{C}_j$, 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 the contract $\mathcal{C}_{lMeter}$.

As also shown in Figure 3, 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. Note that the following definition considers an extension [78] of the one presented in [79] where this extension relaxes the constraint that there must exist a contract for each element in the architecture. However, the definition in the present paper

is slightly simplified in comparison to the definition of a contract structure in [78] where top-level assumptions do not need to be allocated. However, in practice, this simplification is without loss of generality since these top-level assumptions can all be allocated to a single 'environment' element with an interface that is the union of the port variables constrained by the top-level assumptions.

Definition 5 (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 Section 2.5.1 and as also shown in Figure 3, conditions (a) and (b) of Definition 5 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 5 and shown in Figure 3, each node in a contract structure corresponds to a requirement

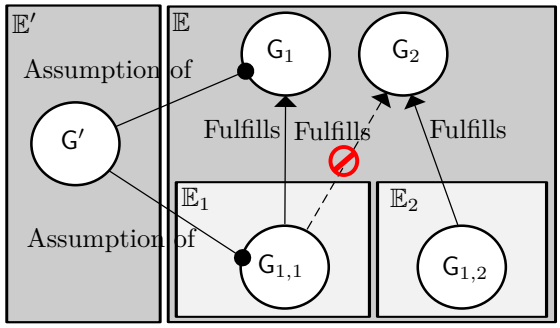


Figure 4. A contract structure that would not be a contract structure if the dashed arc is added to the graph.

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 Figure 4 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 5. In Figure 4, the intent is that the guarantee G_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 G_2 is to be fulfilled by the behavior of $\mathbb{E}$, *given that its environment fulfills* G'. This is due to the fact that the graph also specifies that $\mathsf{G}_{1,1}$ is to, together with $\mathsf{G}_{1,2}$, fulfill G_2, and the behavior of the child $\mathbb{E}_1$ of $\mathbb{E}$ is to fulfill $\mathsf{G}_{1,1}$, given that G' is fulfilled.

For a more detailed explanation of the concept of contract structures, see [78] where this concept is also applied in a major industrial case study. The concept of contract structures recasts the notion of decomposition structures as presented in [77] in the context of RE concepts and extends the notion from two levels to an arbitrary number. Contract structures have a lot in common with Goal Oriented Requirements Engineering (GORE) models, see e.g. I* [82] or KAOS [51] or [52] for a survey, where [51, 82] draw on ideas presented in [43, 44, 63]. The main conceptual difference is that GORE models are more specific since the use of assumptions, also called *expectations*, cannot be used at lower levels and are strictly limited to constrain the environment of a *SW* system. Furthermore, a similar concept

to a contract structure is presented in [61, 62] based on Bayesian networks, but with the specific focus to model failure propagation.

2.5.3 Sufficient Conditions on Requirements in Contract Structure

This section presents a theorem based on a contract structure where the theorem expresses sufficient conditions of property (3). This theorem, which now follows, corresponds to a theorem presented in [78].

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* $(\mathcal{A}, \mathsf{G}, X) \in \mathfrak{C}$ *for* $\mathbb{E} = (X', \mathsf{B})$ *if:*

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

ii) it holds that

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

$$X_{\mathsf{G}} \subseteq X_{Env_{\mathscr{A}}(\mathbb{E})} \cup X \ , \tag{5}$$

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

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

See [78] for a proof of the concepts in Theorem 1.

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 G 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 G, do in fact hold.

Condition (ii) embeds condition (4), which corresponds to condition (2) presented in Section 2.4, and also condition (5). Figure 5 shows an example where condition 5 is violated with respect to a contract $\mathcal{C}''_{Esys} =$

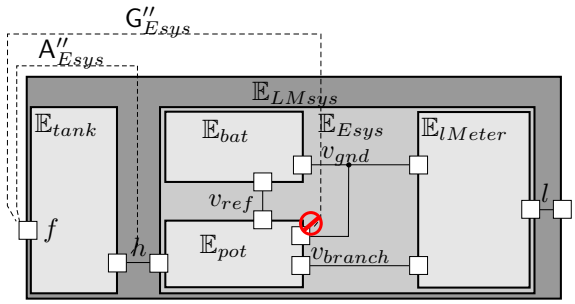


Figure 5. A contract $\mathcal{C}''_{Esys}$ where condition 5 is violated.

$(\{\mathsf{A}''_{Esys}\}, \mathsf{G}''_{Esys}, X_{Esys})$ for the element $\mathbb{E}_{Esys}$ in the architecture previously described in Section 2.3. In this example, condition 5 is violated due to the fact that the guarantee G''_{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 that $X_{\mathsf{G}''_{Esys}} \not\subseteq X_{tank} \cup X_{Esys}$.

Now, to illustrate the use of Theorem 1, consider the contract structure shown in Figure 3. Since $X_{\mathsf{G}_{LMsys}} \subseteq X_{LMsys}$ and since the relations $\mathsf{G}_{lMeter} \subseteq \mathsf{G}_{Esys}$ and $\mathsf{G}_{Esys} \subseteq \mathsf{G}_{LMsys}$ hold, it can be inferred, through the use of Theorem 1, that if the leaf elements of the architecture in Figure 2b satisfy their contracts, then $\mathbb{E}_{LMsys}$ satisfies $\mathcal{C}_{LMsys}$.

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 [3] or the timing model [56]. See also [55, 59] for more discussions on such matters. □

2.6 Extending Contracts Theory with SILs

As described in Section 2.5, 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. Based on a contract

structure, the work in [79] presents formal definitions for how to assign SILs for requirements in a contract structure in accordance with FuSa standards.

Due to space limitations, these formal definitions of SIL assignment will not be presented in the present paper. However, Section 3 will describe how these definitions can be used to also support the assignment of SILs to requirements organized as a contract structure.

3 Tool Support for Specification by Enforcing Syntactic Contract Conditions

Section 2 presented a contract theory for safety-critical heterogeneous system. In particular, this contracts theory includes Theorem 1 that can, in combination with Definition 5, i.e. the definition of a contract structure for an architecture, be used for establishing completeness between requirements on different levels in a hierarchy. Thus, this definition and theorem formalize the particularly stringent RE advocated in FuSa standards such as IEC 61508 and ISO 26262.

Theorem 1 and Definition 5 sets the basis for describing the main contribution, which is presented in this section. More specifically, as will be shown in this section, the conditions of Definition 5 and Theorem 1 can be divided into conditions that can and cannot be enforced when architectures and requirements are not represented formally, but only semi-formally; the former conditions are called *syntactic contract conditions*. After identifying these syntactic contract conditions, it is shown how a specification tool can provide *feedback*- and *guidance*-driven support for authoring specifications by enforcing these syntactic contract conditions. Notably, such support is relevant for heterogeneous systems developers in general, and for complying with the stringent RE advocated by FuSa standards, in particular.

Prior to identifying the syntactic contract conditions and presenting the feedback- and guidance-driven support that can be provided by enforcing these conditions, it will be described how specifications can be structured to be able to specify contract structures for architectures.

3.1 Structuring Specifications in Specification Tool

Consider a specification tool where each specification S_i from a global set $\{S_i\}_{i=1}^{N}$ is structured as an ordered set:

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

- $Alloc_i$ is a variable, called *the allocation of* S_i, that is either NIL or equal to an element;
- X_i 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 $\{(\mathcal{A}_{i,1}, \mathsf{R}_{i,1}, X_i), \ldots, (\mathcal{A}_{i,N_i}, \mathsf{R}_{i,N_i}, X_i)\}$ 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, in the following, that the specification tool ensures that this graph is a DAG by prohibiting arcs to be included in $Assu_{i,j}$ and $Full_{i,j}$ if these arcs result in cyclic dependencies. It follows that this graph is in accordance with the conditions (i)-(iii) of Definition 5, i.e. the definition of a contract structure. That is, the specification structure in itself enforces these conditions.

3.2 Identifying syntactic contract conditions

Given an architecture $\mathscr{A}$, consider the effort of establishing completeness between each requirement level specified by $\{S_i\}_{i=1}^{N}$ in accordance with property 3. That is the effort of establishing that $\{S_i\}_{i=1}^{N}$ is such that Theorem 1 can be used to show that an arbitrary contract $\mathcal{C}$, specified by a specification

in $\{S_i\}_{i=1}^{N}$, is satisfied by the allocation $\mathbb{E}$ of the specification if condition (i) of Theorem 1 holds, i.e. if

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

In accordance with the specification structure described in Section 3.1, Definition 5, and Theorem 1, this effort consists of tasks (I)-(IV) described in Section 1, i.e. the tasks of specifying:

I) *"allocation of requirements to architecture elements"*, i.e. specifying each All_i such that:

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

II) *"interfaces of architecture elements"*, i.e. specifying each X_i to be in accordance with condition (4) (of condition (ii) of Theorem 1):

$$X_i = X_i' \text{ if } Alloc_i = (X_i', \mathsf{B}) \ ;$$

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

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

$$X_{\mathsf{R}_{i,j}} \subseteq X_{Env_{\mathfrak{C}}(\mathbb{E}_i)} \cup X_i' \ ; \text{ and}$$

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

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

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

IV) *"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 5, 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 5, 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 $\mathsf{R}_{i,j}$.

Now, first assume that the architecture $\mathscr{A}$ is represented such that it can only be determined that condition (a)-(i) and (b) of Definition 3 hold and not condition (a)-(ii). That is, the representation is not formal, but *semi-formal* where it is possible to distinguish the hierarchical structuring of elements and their interfaces, but not their behaviors. In the following, data that describes such a representation will be called *non-behavioral architecture data*.

Second, assume that each requirement $\mathsf{R}_{i,j}$ is, rather than being represented formally, represented *semi-formally* as free text embedding distinguishable port variables. For example, the requirement R_{LMsys}, shown in Figure 3, can be represented semi-formally as:

> *'level* **l**[%], *presented by the level meter, shall be equal to actual level* **f**[%] *in the tank'.*

where **l**[%] and **f**[%] are port variables.

Given these two assumptions, out of all the conditions associated with tasks (I)-(IV) presented above, the only condition that cannot be evaluated, is condition (iii) of Theorem 1. The other conditions are indeed syntactic since they can be evaluated without explicitly considering the runs that are in element behaviors or requirements. These insights are summarized in Table 1.

To exemplify the interpretation of Table 1, consider formal condition (6): $Alloc_i = (X_i', \mathsf{B})$ where $(X_i', \mathsf{B}) \in \mathscr{A}$. This condition is associated with

Table 1. Formal interpretations of tasks (I)-(IV) in syntactical and non-syntactical conditions.

Task	Formal Condition(s)	Syntactic?
(I)	Condition (6)	Yes
(II)	Condition (4) (of condition (ii) of Theorem 1)	Yes
(III)	Condition (5) (of condition (ii) of Theorem 1)	Yes
	Condition (iii) of Theorem 1	No
(IV)	Conditions (a)-(b) and (iv)-(v) of Definition 5	Yes

task (I) and expresses that $Alloc_i$ shall be equal to an element in the given architecture $\mathscr{A}$. Thus, this condition is *violated* if $Alloc_i$ is either NIL or equal to an element that is not in $\mathscr{A}$.

3.3 Feedback- and Guidance-Driven Support for Tasks (I)-(IV)

Given non-behavioral architecture data and a specification tool as presented in Section 3.1, this section presents the main contribution that describes how such a specification tool provides support for each of the tasks (I)-(IV) by enforcing the syntactic contract conditions in Table 1. This support is considered to be of two different types, *feedback* and *guidance*.

Feedback is considered to include the cases where the specification tool notifies a user that data in a specification violates one of the syntactic contract conditions. *Guidance* covers the cases where the specification tool distinguishes to the user, prior to inserting data, that certain data is in accordance with these syntactic contract conditions and some data is not.

Note that the violation of a syntactic contract condition in a specification can, in special cases, be identified even prior to specifying an element to be the allocation of the specification. However, the focus in the rest of this section will be on being complete in describing the support that is provided for task (II)-(IV) whenever the allocation of the specification has been specified to be equal to an element. Violations that can be identified whenever the allocation of the specification is NIL, will in any case be identified when an element is eventually specified to be the allocation of the specification. In the following, if the allocation of the specification is equal to an element, then the specification will be said to be *allocated to this element*.

3.3.1 Task (I) – Specifying Allocation of Requirements to Architecture Elements

As shown in Table 1 and as previously mentioned in Section 3.2, the formal interpretation of task (I) is a syntactic contract condition. This means that the tool will notify a user whenever a specification has not been allocated yet and give feedback if the specification has been allocated to an element that is not in the non-behavioral architecture data.

Consider allocating a specification to an element (X, B) in the non-behavioral architecture data. Notably, in accordance with the syntactic contract conditions in Table 1, it is possible that certain conditions can be evaluated *directly* after the allocation, but not prior to the allocation.

This means that, whenever the specification has not been allocated, the specification tool will guide the user in allocating the specification by distinguishing to the user between which elements (in the non-behavioral architecture data) that the specification can and cannot be allocated 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 Figure 3 except that the requirement R_{lMeter} has not yet been allocated. That is, the requirement R_{lMeter} is in a specification that has not yet been allocated to a specific element. Prior to allocating this specification, the specification tool will distinguish to the user that if this specification is allocated to e.g. $\mathbb{E}_{Esys}$ instead of $\mathbb{E}_{lMeter}$, then conditions (a)-(b) of Defintion 5 will be violated since $\mathbb{E}_{Esys}$ is not in the environment of $\mathbb{E}_{lMeter}$ in this architecture. □

In Example 1, the specification tool was shown to provide guidance when allocating a specification by distinguishing to the user that conditions (a)-(b) of Defintion 5 would be violated if the specification were to be allocated to a specific element. However, more generally, the specification tool will not only provide guidance by considering the violation of conditions (a)-(b), but rather the violation of all of the syntactic contract conditions associated with tasks (II)-(IV) in Table 1.

3.3.2 Task (II) – Specifying Architecture Element Interfaces Consisting of Port Variables

Similar to task (I) and as shown in Table 1, condition (4) associated with task (II) is a syntactic contract condition. This means that, whenever a specification has been allocated to an element in the non-behavioral architecture data, the specification tool will enforce that the interface of this element is equal to the interface-specifying set in the specification. An example now follows.

Example 2. Consider that a specification contains an interface-specifying set including only the port variable f and that this specification is allocated to the element $\mathbb{E}_{LMsys}$, shown in Figure 2b. The specification tool will give the feedback that the interface of $\mathbb{E}_{LMsys}$ and the interface-specifying set are not equal; the specification tool will also guide the user by distinguishing which port variables that must be added/removed to the interface-specifying set to ensure that they are equal. □

As shown in Example 2, the specification tool will provide guidance and feedback for task (II). Note that the specification tool does not consider neither the interface-specifying set of a specification nor the interface, of the element to which the specification is allocated to in the non-behavioral architecture data, to be the source of correctness; the specification tool will simply recognize if they are not equal. 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 4.

3.3.3 Task (III) – Specifying Requirements

In accordance with Table 1, condition (5) is the only syntactic contract condition associated with task (III). In order to evaluate condition (5), requirements must be represented semi-formally or formally over distinguishable port variables.

Whenever a specification has been allocated to an element, the specification tool will give feedback by identifying each distinguishable port variable in requirements contained in the specification where such a port variable vi-

olates condition (5). An example of when condition (5) is violated is shown in Figure 5.

Furthermore, when specifying requirements in a specification that has been allocated to an element, the tool will guide a user by distinguishing between port variables that a requirement can and cannot be specified over in order for condition (5) to hold. An example of this now follows.

Example 3. Consider specifying a requirement in a specification allocated to the element $\mathbb{E}_{Esys}$ in the architecture shown in Figure 2b. The specification tool will distinguish to the user that specifying the requirement over e.g. the set $\{f, v_{branch}\}$ will violate condition (5), but over $\{f, l, h\}$ it will not. □

3.3.4 Task (IV) – Specifying Trace Links Between Requirements

As shown in Table 1, conditions (a), (b), (iv), and (v) of Definition 5 can be enforced without formal representations of architectures and requirements; in fact, these conditions can be enforced *regardless of the representation format of the requirements.*

Thus, when specifying 'Assumption of' and 'Fulfills' trace links to and from requirements in a specification, the specification tool will guide the user by distinguishing between trace links that will and will not violate conditions (a), (b), (iv), and (v). Furthermore, the tool will give feedback to a user if already established trace links violate these conditions. Examples of when these conditions are violated are presented in the context of an industrial example system in Section 4.5.

3.4 Additional Condition-Enforcing Support

Section 3.3 described the feedback- and guidance-driven support that is provided for tasks (I)-(IV) by a specification tool that enforces the syntactic contract conditions in Table 1. This section briefly discusses other types of support that can be provided by enforcing additional conditions from the theory described in Section 2.

Notably, out of the conditions in Table 1, condition (iii) of Theorem 1 was the only condition that was not discussed in Section 3.3 since this condition cannot be enforced unless requirements are represented formally. However, if requirements are indeed represented formally, e.g. in the language

TLA+ [50], then an approach such as [83] can be used to enforce condition (iii) of Theorem 1 in a specification tool as described in Section 3.3.

As previously mentioned in Section 2.6, the work in [79] presents formal definitions of SIL assignment given a contract structure for an architecture. Thus, the specification tool can also provide support for SIL assignment by enforcing these definitions, which can also be evaluated regardless of the representation format of requirements and behaviors of elements in the architecture. Without getting into specifics, an example of where these definitions would be violated is if the requirement R_{Esys}, shown in the contract structure in Figure 3, is assigned with a lower SIL than the requirement R_{LMsys}.

4 Design and Implementation of the Specification Tool in an Industrial Setting

This section describes, as the second contribution and validation of the main contribution, how the feedback- and guidance-driven support described in Section 3 can be implemented in an industrial setting. This is achieved by describing the specification tool implemented in the validating industrial case study where this support was realized in the context of an industrial tool chain at Scania. The basic design of the specification tool is first described and is then continuously refined with more implementation specific details.

However, prior to describing the design and implementation of the specification tool, an example system that was used in the industrial case study will be introduced. This industrial example will in the following be used instead of the LM-system for illustrating concepts. The LM system has, due to its simplicity, served as a suitable example for introducing formal concepts; however, it is not a representative industrial system and is not, hence, ideal for illustrating practical application of concepts.

4.1 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. FLD will be described in terms of an architecture and a contracts structure for this architecture as shown in Figure 6 and 7, respectively.

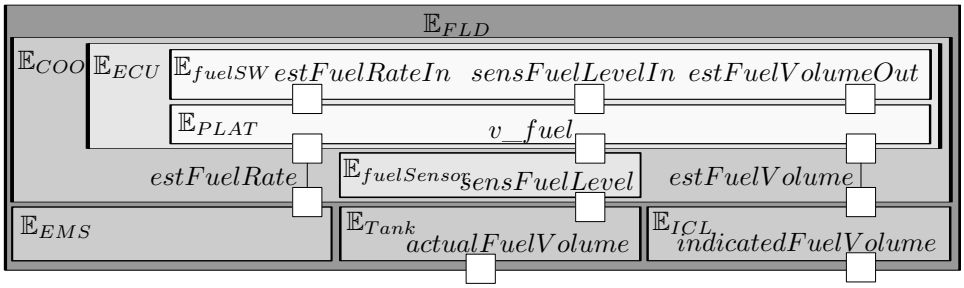


Figure 6. Architecture of FLD.

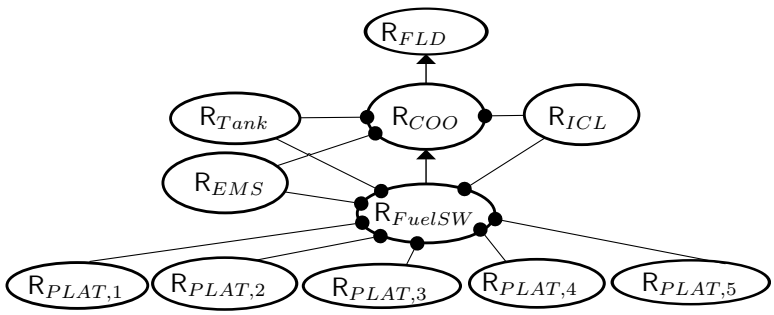


Figure 7. Safety requirements in a contract structure for the architecture shown in Figure 6.

4.1.1 FLD Architecture

As shown in Figure 6, FLD $\mathbb{E}_{FLD}$ consists of a fuel tank $\mathbb{E}_{Tank}$ and three ECU (Electric Control Unit)-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 [78] 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 $estFuelRateIn[l/h]$ and

$sensFuelLevelIn$[%] and output $estFuelVolumeOut$[%] to $\mathbb{E}_{FuelSW}$ correspond to the inputs $estFuelRate[l/h]$ 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 $estFuelRate[l/h]$ 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.

4.1.2 Contract Structure for FLD Architecture

Each requirement shown in Figure 7 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 Section 2.1, represented *semi-formally* 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 semi-formally 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 semi-formally as

> '**indicatedFuelVolume**[%] *corresponds to* **estFuelVolume**[%]'.

4.2 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 Section 3.1, i.e. the allocation of the specification, its interface-specifying set, requirements, and their trace links. Other than data contained in a specification, data in e.g. other specifications, can be presented when opening the specification in the specification tool. This

is achieved through *referencing* this other data in accordance with Linked Data [12].

That is, inserting a reference means inserting a Uniform Resource Identifier (URI) that is associated with the data that is to be presented. However, the URI does not only contain the data that is to be presented, but also information on how the data should be interpreted, i.e. *dereferenced*. Inserted URIs can be dereferenced in a standardized manner using Hypertext Transfer Protocol (HTTP), which means that inserted URIs are actually Uniform Resource Locator(URL)s.

Whenever a URL is inserted, the specification tool will first interpret 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 that sense, the data is *linked*.

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 is *consistent*, i.e. updated with the data that is associated with the URL. Consider the following example.

Example 4. Assume that non-behavioral architecture data in accordance with Figure 6 is made available as Linked Data. Consider inserting an URL associated with the port variable $indicatedFuelVolume[\%]$ with the intent of specifying the requirement R_{FLD} semi-formally. 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 enables data in specifications to be consistent. This is a crucial property in an industrial setting. Consider that data in specifications is 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 is continuously consistent is unmanageable even for a relatively small set of specifications.

4.3 Using RDF for Publishing and Consuming Linked Data

This section explains how data can be made available as Linked Data in practice.

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. 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; in practice, this typically means using Resource Description Framework (RDF) [47].

The specification tool publishes the data contained in specifications as RDF triples, consisting of a subject (URI), a predicate (URI), and an object (URI or a literal). Consider that this published data is 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 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 non-behavioral architecture data;
- each member in the interface-specifying set where this member is a reference to a port variable in non-behavioral architecture data, corresponds to a link from the interface-specifying set to this port variable in non-behavioral architecture data; and
- the allocation of the specification, whenever containing a reference to an element in non-behavioral 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 [65] for querying data. The specification tool uses SPARQL for evaluating the syntactic contract conditions in Table 1. More specifically, these evaluations

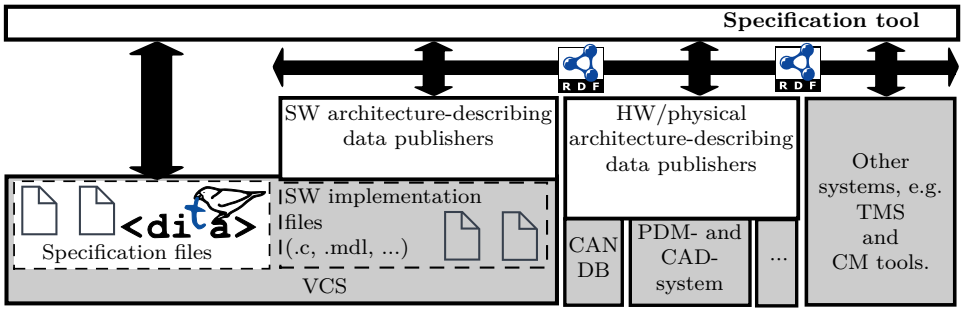


Figure 8. Integration of specification tool into the industrial tool chain at Scania.

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 non-behavioral architecture data has not been described. This will be done in Section 4.4, which follows after this section. However, the fact is that, when using the Linked Data approach, the source of the non-behavioral 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.

Actually, the fact that it is *not* important to specifically identify the source of the non-behavioral architecture data, is crucial in order to realize the support described in Section 3.3 in practice. Consider a large company where development is distributed over several groups using different tools for modeling different parts of the architecture of a system. Evaluating the syntactic contract conditions in Table 1 over the *latest updated* data in *all* of these tools, is indeed infeasible *without* the data being linked and possibly to evaluate in a uniform fashion.

4.4 Integration of Specification Tool into Industrial Tool Chain

The integration of the specification tool into the tool chain at Scania is shown in Figure 8 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 Figure 8, this bidirectional arrow represent 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 XML-files in accordance with Darwin Information Typing Architecture (DITA)[35] technology. 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 [21] .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 Figure 8, capture publishing/consumption of data as discussed in Section 4.3. 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 certain 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 non-behavioral architecture data published by other tools in the tool chain.

Consider published non-behavioral 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.

For example, data extracted from a .h-file associated with a .c-file could be an example of non-behavioral architecture-describing data. An example of architecture-specifying data could be data in a high-level architecture model, represented in a language such as SysML, Unified Modeling Language (UML) [71], or Architecture Analysis and Design Language (AADL) [27]. 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, this data is obtained from tools, i.e. publishers, which automatically analyze and ex-

tract data directly from SW implementation files in VCS using *architecture recovery* [67, 84]. 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 is *consistent* with SW files. In addition, the published architecture-describing data is also linked to version data.

In a similar manner, HW/physical architecture-describing data publishers automatically analyze and extract data from sources that contain data that describes 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 that describes properties of sensors and actuators, Computer Aided Design (CAD)-systems.

4.5 Authoring Support in Specification Tool

With respect to authoring specifications, the specification tool functions similar to a typical text editor, e.g. MS 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 Figure 8, 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 Figure 8.

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 Section 3.3, 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 1 are enforced.

This principle is that the syntactic contract conditions are evaluated with respect to 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 is not available; in an early design phase

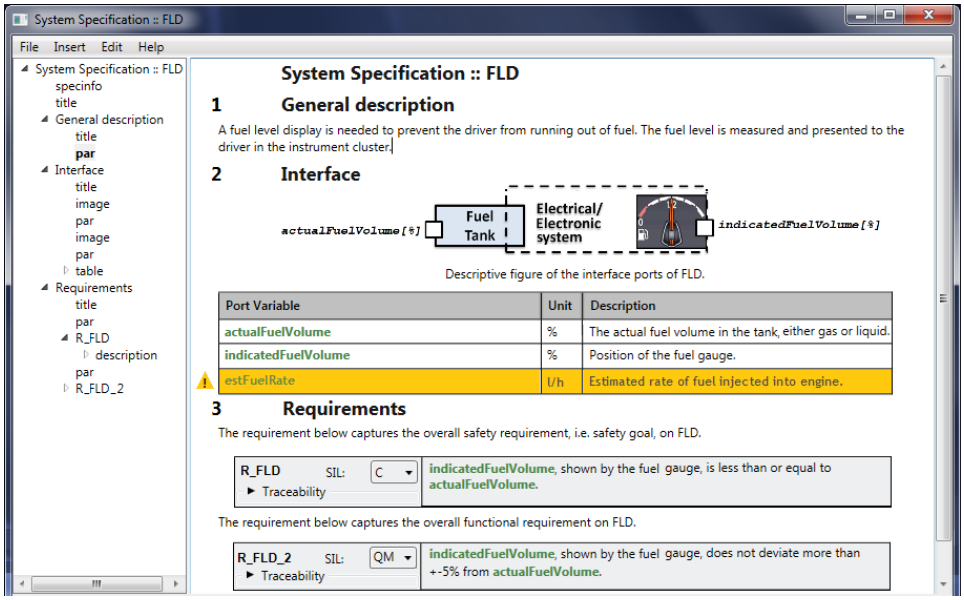


Figure 9. Snapshot of the main window of the UI of the specification tool implemented in the industrial case study.

in particular, it might be the case that only architecture-specifying data is 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, this data should *eventually* be replaced with references to architecture-describing data and the specification tool will therefore continuously alert the user until this is done.

4.5.1 Task (I) – Specifying Allocation of Requirements to Architecture Elements

As explained in Section 3.1, allocating a specification to an element means allocating the requirements in the specification to this element. This declaration 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 specification is to be allocated to. In accordance with Section 3.3.1, when selecting from this list, the specification tool will guide the user by distinguishing between elements (in the non-behavioral architecture data) that the specification can and cannot be allocated to without violating the syntactic contract conditions associated with tasks (II)-(IV).

Allocating a specification to an element can be done at any time after creating the specification. Prior to allocating the specification, the tool will continuously urge the user to allocate the specification.

Considering the machine-readable structure of the specification and its specification file, selecting an element from this list means that a reference to this element is stored somewhere in this structure. When publishing data about the specification, each of its contained requirements are linked to this element in the architecture-describing data.

4.5.2 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 is a special table specifying 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 Figure 9.

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 Figure 10 when entering data into the table also shown in Figure 9. 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-

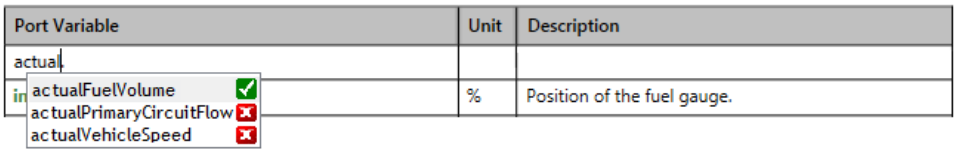


Figure 10. Entering data into interface table using auto-complete functionality.

describing data, but also other relevant properties.

Whenever the specification is allocated to an element $\mathbb{E}$, in accordance with Section 3.3.2, the specification tool will enforce condition (4) and give feedback and guidance. An example is shown in Figure 9 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 is in accordance with Figure 6.

Notably, as shown in Figure 10, 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 (4). For example, in Figure 10, the port variables that are and are not in accordance with condition (4) 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 the architecture-describing data in accordance with condition (4), thus ensuring consistency and saving much manual and error-prone work.

4.5.3 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. In addition, a SIL can be optionally specified for the requirement; SILs are discussed further in Section 4.5.5.

Explicit support is given for specifying requirements in semi-formal representation as free text with references to port variables. Two examples of such a representation format was presented in Section 4.1.2.

This support for specifying requirements in semi-formal representation is

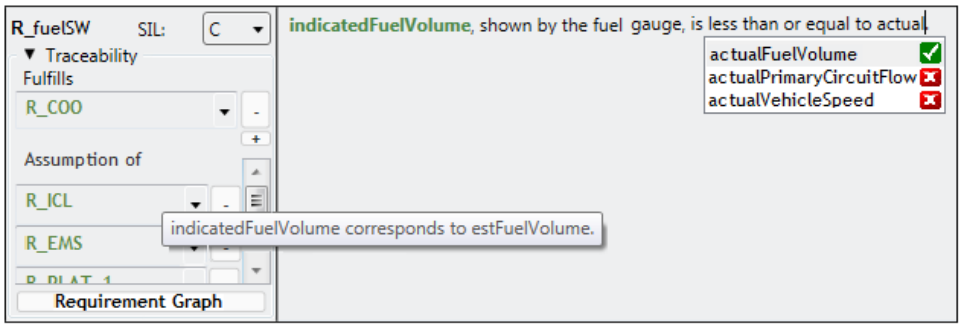


Figure 11. Snapshot of a requirement in the specification tool implemented in the industrial case study.

given by having auto-complete functionality for entering port variable references. Figure 11 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), 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 (5).

In accordance with Section 3.3.3, whenever a specification has been allocated to an element, the specification tool will also give feedback on requirements specified prior to their allocation. That is, the tool will flag port variable references that violate condition (5).

4.5.4 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 exemplified in Figure 11, 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 Figure 11 for the requirement R_{ICL}.

Considering the machine-readable structure of the specification and its specification file, selecting an 'Assumption of' trace link source or a 'Fulfills' trace link target for R means that a reference to this source/target require-

ment is inserted and associated with R. When data on the specification is published, this association is captured as a link between these requirements. As also shown in Figure 11, there is also an option to view published data on requirements and their links as a navigable 'Requirements Graph' (similar to Figure 7) that shows R in the context of neighboring requirements that are traceable from or to R through specified requirement trace links.

That is, in accordance with Section 3.3.4, whenever a specification has been allocated to an architecture element $\mathbb{E}$, 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) of Definition 5. Each of these conditions can be enforced regardless of the representation format of requirements. Examples of each of these conditions now follow.

As a first example of how the specification tool can provide guidance, consider a specification that contains R_{COO} and that is allocated to $\mathbb{E}_{COO}$. Consider also that all the requirements in the graph shown in Figure 7 are allocated as described in Section 4.1.2. 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 distinguishable as not being in accordance with 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 Figure 11. 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) of Definition 5. 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 (iv) of Definition 5, enforcing condition (v) also allows distinguishing between the selection of possible 'Fulfills' trace link targets. An example of a case were adding a 'Fulfills' trace link would violate condition (v) was previously presented in Section 2.5.2 and is shown in Figure 4.

Note that the specification tool will allow any trace links to be specified as long as these do not result in cyclic dependencies, even if these trace links

result in that syntactic contract conditions are violated. However, whenever specified trace links result in that syntactic contract conditions are violated, the tool will provide feedback by notifying the user and flagging these trace links.

4.5.5 Condition-Enforcing Support for Specifying SILs

As previously mentioned in Section 3.3.3, SILs can be specified for requirements in a specification. The specification tool enforces the definitions of SIL assignment that are presented in [79] 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 Figure 7 is assigned with a lower SIL than the requirement R_{FLD}.

5 Conclusion

This paper has presented tool support for authoring structured specifications for heterogeneous systems. More specifically and as the main contribution, by having a specification tool enforcing syntactic conditions from established contracts theory, it has been shown how guidance- and feedback-driven support can be given for tasks (I)-(IV), which constitute the basis of the stringent RE effort advocated by FuSa standards such as IEC 61508 and ISO 26262.

It has been shown that such support can be given for structuring requirements, regardless of their representation format. In contrast, in order to provide feedback and guidance when specifying requirements, the requirements must be expressed over references to port variables in an architecture describing the implemented system. However, 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 principal concept for enabling such analyzes in practice is Linked Data, which supports formal referencing and dereferencing of data in between specifications and non-behavioral 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 this data. Moreover, if architecture-describing data is 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 desired property when attempting to achieve FuSa, but rather a desired 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.

References

[1] Def Stan 00-56: Safety management requirements for defence systems (2007)

[2] Abadi, M., Lamport, L.: The Existence of Refinement Mappings. Theor. Comput. Sci. **82**(2), 253–284 (1991). doi:10.1016/0304-3975(91)90224-P. URL http://dx.doi.org/10.1016/0304-3975(91)90224-P

[3] Abadi, M., Lamport, L.: Composing specifications. ACM Trans. Program. Lang. Syst. **15**(1), 73–132 (1993). doi:10.1145/151646.151649. URL http://doi.acm.org/10.1145/151646.151649

[4] Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. International journal on software tools for technology transfer **12**(6), 447–466 (2010)

[5] Almeida, J.P.A., Iacob, M.E., Van Eck, P.: Requirements traceability in model-driven development: Applying model and transformation conformance. Information Systems Frontiers **9**(4), 327–342 (2007)

[6] Alur, R., Henzinger, T.: Reactive Modules. Formal Methods in System Design **15**(1), 7–48 (1999). doi:10.1023/A:1008739929481. URL http://dx.doi.org/10.1023/A%3A1008739929481

[7] Bauer, S., David, A., Hennicker, R., Guldstrand Larsen, K., Legay, A., Nyman, U., Wąsowski, A.: Moving from specifications to contracts in component-based design. In: J. Lara, A. Zisman (eds.) Fundamental Approaches to Software Eng., *Lec. Notes in Computer Science*, vol. 7212, pp. 43–58. Springer Berlin Heidelberg (2012). doi: 10.1007/978-3-642-28872-2$_$3. URL http://dx.doi.org/10.1007/978-3-642-28872-2$_$3

[8] Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple Viewpoint Contract-Based Specification and Design. In: F. de Boer, M. Bonsangue, S. Graf, W.P. de Roever (eds.) Formal Methods for Components and Objects, *Lecture Notes in Computer Science*, vol. 5382, pp. 200–225. Springer Berlin Heidelberg (2008). doi: 10.1007/978-3-540-92188-2_9. URL http://dx.doi.org/10.1007/978-3-540-92188-2_9

[9] Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.G.: Contracts for System Design. Rapport de recherche RR-8147, INRIA (2012). URL http://hal.inria.fr/hal-00757488

[10] Benveniste, A., Caillaud, B., Passerone, R.: A Generic Model of Contracts for Embedded Systems. Research Report RR-6214, INRIA (2007). URL http://hal.inria.fr/inria-00153477

[11] Benveniste, A., Caillaud, B., Passerone, R.: Multi-Viewpoint State Machines for Rich Component Models. In: G. Nicolescu, P. Mosterman (eds.) Model-Based Design for Embedded Systems, pp. 487–518. Taylor & Francis (2009). URL http://www.google.se/books?id=8Cjg2mM-m1MC

[12] Bizer, C., Heath, T., Berners-Lee, T.: Linked data-the story so far. Semantic Services, Interoperability and Web Applications: Emerging Concepts pp. 205–227 (2009)

[13] Böschen, M., Bogusch, R., Fraga, A., Rudat, C.: 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 (2016). URL http://ceur--ws.org/Vol--1564/paper20.pdf

[14] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A Theory of Communicating Sequential Processes. J. ACM **31**(3), 560–599 (1984). doi:10.1145/828.833. URL http://doi.acm.org/10.1145/828.833

[15] Cheng, B.H.C., Atlee, J.M.: Research directions in requirements engineering. In: Future of Software Engineering, 2007. FOSE '07, pp. 285–303 (2007). doi:10.1109/FOSE.2007.17

[16] Chilton, C., Jonsson, B., Kwiatkowska, M.: An algebraic theory of interface automata. Theoretical Computer Science **549**, 146–174 (2014)

[17] Cimatti, A., Dorigatti, M., Tonetta, S.: Ocra: A tool for checking the refinement of temporal contracts. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pp. 702–705 (2013). doi:10.1109/ASE.2013.6693137

[18] Cimatti, A., Tonetta, S.: Contracts-refinement proof system for component-based embedded systems. Science of Computer Programming **97, Part 3**, 333 – 348 (2015). doi:http://dx.doi.org/10.1016/j.scico.2014.06.011. URL http://www.sciencedirect.com/science/article/pii/S0167642314002901. Object-Oriented Programming and Systems (OOPS 2010)Modeling and Analysis of Compositional Software (papers from {EUROMICRO} SEAA'12)

[19] Cleland-Huang, J., Gotel, O., Zisman, A.: Software and Systems Traceability. Springer Publishing Company, Incorporated (2012)

[20] Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: Proceedings of the 4th International Conference on NASA Formal Methods, NFM'12, pp. 126–140. Springer-Verlag, Berlin, Heidelberg (2012). doi:10.1007/978-3-642-28891-3$_$13. URL http://dx.doi.org/10.1007/978-3-642-28891-3$_$13

[21] Dabney, J.B., Harman, T.L.: Mastering simulink. Pearson/Prentice Hall (2004)

[22] Damm, W.: Controlling Speculative Design Processes Using Rich Component Models. In: Application of Concurrency to System Design, 2005. ACSD 2005. Fifth International Conference on, pp. 118 – 119 (2005). doi:10.1109/ACSD.2005.35

[23] Diestel, R.: Graph Theory, 4th Edition, *Graduate texts in mathematics*, vol. 173. Springer (2012)

[24] Dill, D.L.: Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. In: Proceedings of the fifth MIT conference on Advanced research in VLSI, pp. 51–65. MIT Press, Cambridge, MA, USA (1988). URL http://dl.acm.org/citation.cfm?id=88056.88061

[25] EN 50128: Railway applications - communication, signalling and processing systems - software for railway control and protection systems (2011)

[26] Farfeleder, S., Moser, T., Krall, A., Stålhane, T., Zojer, H., Panis, C.: DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development. In: Design and Diagnostics of Electronic Circuits Systems (DDECS), 2011 IEEE 14th International Symposium on, pp. 271–274 (2011). doi:10.1109/DDECS.2011.5783092

[27] Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, 1st edn. Addison-Wesley Professional (2012)

[28] Firesmith, D.: Engineering safety requirements, safety constraints, and safety-critical requirements. Journal of Object technology **3**(3), 27–42 (2004)

[29] Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: Systems Modeling Language. Morgan Kaufmann Inc., San Francisco, CA, USA (2008)

[30] Fritzson, P.: Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach. John Wiley & Sons (2014)

[31] Fritzson, P., Engelson, V.: Modelica - A unified object-oriented language for system modeling and simulation. In: E. Jul (ed.) ECOO'98 - Object-Oriented Programming, *Lecture Notes in Computer Science*, vol. 1445, pp. 67–90. Springer Berlin Heidelberg (1998). doi:10.1007/BFb0054087. URL http://dx.doi.org/10.1007/BFb0054087

[32] Génova, G., Fuentes, J.M., Llorens, J., Hurtado, O., Moreno, V.: A framework to measure and improve the quality of textual requirements. Requirements Engineering **18**(1), 25–41 (2013). doi:10.1007/s00766-011-0134-z. URL http://dx.doi.org/10.1007/s00766-011-0134-z

[33] Goknil, A., Kurtev, I., Van Den Berg, K.: Generation and Validation of Traces Between Requirements and Architecture Based on Formal Trace Semantics. J. Syst. Softw. **88**(C), 112–137 (2014). doi:10.1016/j.jss.2013.10.006. URL http://dx.doi.org/10.1016/j.jss.2013.10.006

[34] Hähnle, R., Johannisson, K., Ranta, A.: 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-Verlag, London, UK, UK (2002). URL http://dl.acm.org/citation.cfm?id=645370.651289

[35] Harrison, N.: The Darwin information typing architecture (DITA): Applications for globalization. In: IPCC 2005. Proceedings. International Professional Communication Conference, 2005., pp. 115–121. IEEE (2005)

[36] Henzinger, T., Sifakis, J.: The Discipline of Embedded Systems Design. Computer **40**(10), 32–40 (2007). doi:10.1109/MC.2007.364

[37] Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Commun. ACM **12**(10), 576–580 (1969). doi:10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259

[38] Hooman, J., de Roever, W.P.: The Quest Goes on: A Survey of Proofsystems for Partial Correctness of CSP. In: Current Trends in Concurrency, Overviews and Tutorials, pp. 343–395. Springer Berlin Heidelberg (1986). doi:10.1007/BFb0027044. URL http://dx.doi.org/10.1007/BFb0027044

[39] Hull, M.E.C., Jackson, K., Dick, J. (eds.): Requirements Engineering, Third Edition. Springer (2011)

[40] IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems (2010)

[41] ISO 26262: Road vehicles-Functional safety (2011)

[42] ISO/IEC/IEEE 42010: System and software eng. - Architecture description (2011)

[43] Jackson, M.: Software Requirements &Amp; Specifications: A Lexicon of Practice, Principles and Prejudices. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (1995)

[44] Jackson, M.: The world and the machine. In: Proceedings of the 17th International Conference on Software Engineering, ICSE '95, pp. 283–292. ACM, New York, NY, USA (1995). doi:10.1145/225014.225041. URL http://doi.acm.org/10.1145/225014.225041

[45] Jones, C.B.: Specification and Design of (Parallel) Programs. In: R.E.A. Mason (ed.) Information Processing 83, *IFIP Congress Series*, vol. 9, pp. 321–332. IFIP, North-Holland, Paris, France (1983)

[46] Josko, B., Ma, Q., Metzner, A.: Designing Embedded Systems using Heterogeneous Rich Components. In: Proceedings of the INCOSE International Symposium (2008)

[47] Klyne, G., Carroll, J.J.: Resource description framework (rdf): Concepts and abstract syntax (2006)

[48] Knauss, E., Lubke, D., Meyer, S.: Feedback-driven requirements engineering: The Heuristic Requirements Assistant. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 587–590 (2009). doi:10.1109/ICSE.2009.5070562

[49] Lamport, L.: A Simple Approach to Specifying Concurrent Systems. Commun. ACM **32**(1), 32–45 (1989). doi:10.1145/63238.63240. URL http://doi.acm.org/10.1145/63238.63240

[50] Lamport, L.: Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co., Inc. (2002)

[51] van Lamsweerde, A., Letier, E.: From object orientation to goal orientation: A paradigm shift for requirements engineering. In: M. Wirsing, A. Knapp, S. Balsamo (eds.) Radical Innovations of Software and Systems Engineering in the Future, *Lecture Notes in Computer Science*, vol. 2941, pp. 325–340. Springer Berlin Heidelberg (2004). doi:10.1007/978-3-540-24626-8_23. URL http://dx.doi.org/10.1007/978-3-540-24626-8_23

[52] Lapouchnian, A.: Goal-oriented requirements engineering: An overview of the current research. University of Toronto (2005)

[53] Lee, E.: Cyber Physical Systems: Design Challenges. In: Object Oriented Real-Time Distributed Computing (ISORC), 11th IEEE Int. Symp. on, pp. 363–369 (2008). doi:10.1109/ISORC.2008.25

[54] Leveson, N.G.: Safeware: System Safety and Computers. ACM, New York, NY, USA (1995)

[55] Maier, P.: Compositional circular assume-guarantee rules cannot be sound and complete. In: A. Gordon (ed.) Foundations of Software Science and Computation Structures, *Lecture Notes in Computer Science*, vol. 2620, pp. 343–357. Springer Berlin Heidelberg (2003). doi:10.1007/3-540-36576-1_22. URL http://dx.doi.org/10.1007/3-540-36576-1_22

[56] Mcmillan, K.L.: 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, pp. 342–345. Springer-Verlag (1999)

[57] Meyer, B.: Applying "Design by Contract". Computer **25**(10), 40–51 (1992). doi:10.1109/2.161279. URL http://dx.doi.org/10.1109/2.161279

[58] Misra, J., Chandy, K.: Proofs of Networks of Processes. Software Engineering, IEEE Transactions on **SE-7**(4), 417–426 (1981). doi:10.1109/TSE.1981.230844

[59] Namjoshi, K.S., Trefler, R.J.: On the completeness of compositional reasoning methods. ACM Trans. Comput. Logic **11**(3), 16:1–16:22 (2010). doi:10.1145/1740582.1740584. URL http://doi.acm.org/10.1145/1740582.1740584

[60] Negulescu, R.: Process Spaces. In: Proceedings of the 11th Int. Conf. on Concurrency Theory, CONCUR '00, pp. 199–213. Springer-Verlag, London, UK, UK (2000). URL http://dl.acm.org/citation.cfm?id=646735.701627

[61] Nyberg, M.: Failure propagation modeling for safety analysis using causal bayesian networks. In: Control and Fault-Tolerant Systems (SysTol), 2013 Conference on, pp. 91–97 (2013). doi:10.1109/SysTol.2013.6693936

[62] Nyberg, M., Westman, J.: Failure Propagation Modeling Based on Contracts Theory. In: Dependable Computing Conference (EDCC), 2015 Eleventh European, pp. 108–119 (2015). doi:10.1109/EDCC.2015.21

[63] Parnas, D.L.: Functional documents for computer systems. Science of Computer Programming **25**, 41–61 (1995)

[64] Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science, 1977., 18th Annual Symposium on, pp. 46–57 (1977). doi:10.1109/SFCS.1977.32

[65] Quilitz, B., Leser, U.: Querying distributed rdf data sources with sparql. In: European Semantic Web Conference, pp. 524–538. Springer (2008)

[66] Quinton, S., Graf, S.: Contract-based verification of hierarchical systems of components. In: Software Engineering and Formal Methods, 2008. SEFM '08. Sixth IEEE International Conference on, pp. 377 –381 (2008). doi:10.1109/SEFM.2008.28

[67] Rasool, G., Asif, N.: Software architecture recovery. International Journal of Computer, Information, and Systems Science, and Engineering **1**(3) (2007)

[68] Rausand, M., Høyland, A.: System Reliability Theory: Models, Statistical Methods, and Applications. Wiley Series in Probability and Statistics - Applied Probability and Statistics Section. Wiley (2004). URL https://books.google.se/books?id=gkUWz9AA-QEC

[69] Rawat, D.B., Rodrigues, J.J., Stojmenovic, I.: Cyber-Physical Systems: From Theory to Practice. CRC Press (2015)

[70] de Roever, W., Langmaack, H., Pnueli, A.: Compositionality: The Significant Difference. Springer (1998)

[71] Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, The (2nd Edition). Pearson Higher Education (2004)

[72] Sangiovanni-Vincentelli, A.L., Damm, W., Passerone, R.: Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. European Journal of Control **18**(3), 217–238 (2012)

[73] Simko, G., Levendovszky, T., Maroti, M., Sztipanovits, J.: Towards a theory for cyber-physical systems modeling. In: Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems, CyPhy '14, pp. 56–61. ACM, New York, NY, USA (2014). doi:10.1145/2593458.2593463. URL http://doi.acm.org/10.1145/2593458.2593463

[74] Sutcliffe, A., Maiden, N.: The domain theory for requirements engineering. IEEE Transactions on Software Engineering **24**(3), 174–196 (1998). doi:10.1109/32.667878

[75] Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)

[76] Westman, J., Nyberg, M.: A Reference Example on the Specification of Safety Requirements using ISO 26262. In: M. ROY (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 (2013). URL http://hal.archives-ouvertes.fr/hal-00848610

[77] Westman, J., Nyberg, M.: Environment-Centric Contracts for Design of Cyber-Physical Systems. In: J. Dingel, W. Schulte, I. Ramos, S. Abrahao, E. Insfran (eds.) Model-Driven Engineering Languages and Systems, *Lecture Notes in Computer Science*, vol. 8767, pp. 218–234. Springer International Publishing (2014). doi:10.1007/978-3-319-11653-2_14

[78] Westman, J., Nyberg, M.: Contracts for Specifying and Structuring Requirements on Cyber-Physical Systems. In: D.B. Rawat, J. Rodriques, I. Stojmenovic (eds.) Cyber Physical Systems: From Theory to Practice. Taylor & Francis (2015)

[79] Westman, J., Nyberg, M.: Extending Contract theory with Safety Integrity Levels. In: HASE, 2015 IEEE 16th Int. Symposium on, pp. 85–92 (2015). doi:10.1109/HASE.2015.21

[80] Westman, J., Nyberg, M., Törngren, M.: 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-Verlag New York, Inc., New York, NY, USA (2013). doi:10.1007/978-3-642-40793-2_16. URL http://dx.doi.org/10.1007/978-3-642-40793-2_16

[81] Wolf, E.S.: Hierarchical models of synchronous circuits for formal verification and substitution. Ph.D. thesis, Stanford University, Stanford, CA, USA (1996). UMI Order No. GAX96-12052

[82] Yu, E.: Towards modelling and reasoning support for early-phase requirements engineering. In: Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on, pp. 226–235 (1997). doi:10.1109/ISRE.1997.566873

[83] Yu, Y., Manolios, P., Lamport, L.: Model checking tla+ specifications. In: Advanced Research Working Conference on Correct Hardware Design and Verification Methods, pp. 54–66. Springer (1999)

[84] Zhang, X., Persson, M., Nyberg, M., Mokhtari, B., Einarson, A., Linder, H., Westman, J., Chen, D., Törngren, M.: Experience on applying software architecture recovery to automotive embedded systems. In: Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on, pp. 379–382. IEEE (2014)