DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2020

Evaluating the utility of a contract compositionality proof framework

ANTON LÖVSTRÖM

ANDERS STEEN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Evaluating the utility of a contract compositionality proof framework

ANTON LÖVSTRÖM

ANDERS STEEN

Bachelor in Computer Science
Date: June 3, 2020
Supervisor: Dilian Gurov
Examiner: Pawel Herman
School of Electrical Engineering and Computer Science
Swedish title: Användbarhetsutvärdering av ett ramverk för korrekthetsbevis inom kontrakt

Abstract

A contract in system design is a concept used for specifying behaviors of and interactions between components in complex systems. Contracts make explicit the mutual commitments between components in a system. Nyberg et al propose a contracts theory and framework for proving the correctness of a decomposition of a system specification. The aim of this project is to evaluate the utility of the framework proposed by Nyberg et al. A decomposition of a concrete system's top-level specification, and a proof of the decomposition's correctness using the framework is performed. Evaluating the utility is done through considering a set of criteria while performing the decomposition and proof. This study finds the contracts theory to be well suited for formalizing natural language specifications and for adapting to systems with differing requirements on detail in specifications. The framework is found to have the problem that the length and complexity of the proofs created using it scale poorly with the size and complexity of the decomposition. However, automating the proof search would render the scaling problem irrelevant. Overall, the contracts theory and framework have high utility in most regards.

Sammanfattning

Kontrakt inom systemdesign är ett koncept som används för att specificera beteenden av och interaktioner mellan komponenter i komplexa system. Kontrakt uttrycker de ömsesidiga ansvaren mellan komponenter i ett system. Nyberg et al föreslår en kontraktteori och ett ramverk för att bevisa korrektheten i dekompositioner av systemspecifikationer. Syftet med denna studie är att evaluera användbarheten av ramverket som Nyberg et al föreslår. En dekomposition av ett konkret systems toppnivå-specifikation, och ett bevis av korrektheten i dekompositionen genomförs. Evalueringen av användbarheten hos ramverket görs genom att betrakta ett antal kriterier som tagits fram för ändamålet. Denna studie finner att kontraktteorin passar bra till att formalisera specifikationer i naturligt språk och går bra att anpassa till system med olika krav på detaljnivån hos specifikationer. Ramverket har visat sig ha problemet att längden och komplexiteten hos de bevis som skapas med hjälp av det skalar dåligt med storleken och komplexiteten för dekompositionen. Automatisering av bevisprocessen skulle emellertid göra skalningsproblemet irrelevant. Sammantaget har kontraktteorin och ramverket hög användbarhet i de flesta avseenden.

Contents

Chapter 1

Introduction

A contract in system design is a concept used for specifying behaviors of and interactions between components in complex systems. The goals in using contracts are facilitating parallel development, integrating subsystems, and avoiding costly corrections late in development. According to [1], contract-based design aims to aid by setting up responsibilities for all involved parties, specifying interfaces between components that are developed independently.

The contract concept was first developed in the context of software engineering, according to [1], with Floyd-Hoare logic as in [2] and Design by Contract as popularized by Bertrand Meyer in [3] in the early 1990s. Using contracts for development of Cyber-Physical Systems became popular later, in the 2000s, as stated by [1]. These two different domains, software engineering and Cyber-Physical Systems, have different contracts theories that have been developed independently of each other. Benveniste et al present a mathematical meta-theory of contracts theories in [1], which defines contracts in a generic and abstract way. Specific contracts theories can be seen as instantiations of this meta-theory.

One contracts theory that could be argued to be an instantiation of the meta-theory is proposed by Nyberg et al in [4]. It defines a theory for creating decompositions of systems' top-level specifications of the system requirements. A system of proof rules for proving the correctness of decompositions is also defined. The framework specifically describes the typical case of non-monotonic logic, making it applicable to many real-life systems, according to Nyberg et al. Despite its applicability, it is defined at a very high level of abstraction, e.g. not explicitly defining a language for specifying component specifications.

This report focuses on Cyber-Physical Systems and the decomposition of a

system's top level specification and requirements, as well as resolving the modeling issues associated with applying an abstract theory on a concrete system. The objective is to evaluate the contracts theory and framework presented in [4] by using it to create decompositions of the system's top-level requirements along its architecture, proving their correctness and through this process evaluating the utility of the contracts theory and proof framework.

1.1 Aim and research question

The abstract contracts theory proposed by Nyberg et al is applied to a concrete example of a system. The top level requirements of the system are decomposed and specified in two different ways, and the decompositions are proven correct using the proof rules and special proof cases in [4]. The focus is on evaluating the utility of the framework through resolving the modeling issues concerning e.g. how to specify component requirements and fixing a time domain on a real example system. These issues comprise a modeling gap between the theory and the actual specifications. Considering the problem of a specification's faithfulness to the real world compared to its complexity is a large part of this. Naturally, there is a certain discrepancy between a real world component and its specification. The more features of a component that are modeled, the more complexity is added to the specification.

The research question could be phrased: *In the contracts theory proposed by Nyberg et al, how do different approaches of solving the modeling problems compare and how can one approach the problem of formalizing natural language specifications into the syntax and semantics of the theory?* No criteria for evaluation of the contracts theory in [4] have been formulated, so part of this study will be to propose the required criteria.

The correctness of the framework proposed by Nyberg et al has been proven in [4], and the aim of this study is to further evaluate the utility of the theory and framework of [4].

1.2 Outline

The background chapter explains the essential concepts in the field of contracts theories. Some history in contracts is covered, followed by more recent and specific contracts theories and finally the non-monotonic contracts theory proposed by Nyberg et al and the concepts on which this report is closely based are explained.

The method chapter specifies and motivates our approach to solving the problem stated, and also presents the criteria for evaluation. The results chapter contains the system decompositions, specifications and proofs obtained through the application of the framework proposed by Nyberg et al. Following the results chapter are the discussion and conclusions chapters, which contain the evaluation of the contracts theory and framework, and also the final conclusions.

Chapter 2

Background

In this chapter, background knowledge required for readers to understand the topic is provided. General information about contracts, certain contracts theories and the specifics of the contracts theory laid forth by Nyberg et al in [4] are presented.

2.1 Contracts

The idea of contracts has been defined in multiple ways in different domains. This section summarizes the concepts on a high level of abstraction, to give an orientation of the field.

2.1.1 Software contracts

The domain where contracts were first defined is that of software contracts. In [3] the notion of these types of contracts is explained. The contracts are explained to be analogous to real-world contracts where there is a client who needs a task to be completed by a contractor. The two parties, client and contractor, make an agreement in the form of a contract on how much should be done and how little is acceptable. The software analogy is that large software systems require different components to do different tasks, and a certain component requires a certain other component to do a certain task. So, a contract is required between the components.

The agreement in the contracts between software components is enforced by assertions in the software, according to [3]. The assertions say what needs to be true about variables and the relationships between variables. A contract is made up of assertions, which take the form of preconditions and postcon-

ditions. The assertions that take the form of preconditions enforce what is required about the variables that are relevant to the task to be performed *before* the task is performed. The postcondition assertions enforce what is guaranteed about the variables that are relevant to the task to be performed *after* the task is performed. These are the definitions provided in [3] about software contracts that are relevant for this report.

An example of a software contract could be that of a modulo function. The precondition states that the numerator should be an integer and that the denominator should be a positive integer. The postcondition says that the function will return the remainder from integer division of the numerator and denominator.

2.1.2 System-level contracts

In [5], contract-based design of cyber-physical systems is discussed. Cyber-physical systems are systems that combine mechanical, electrical or chemical processes with computation and networking. Cyber components often monitor sensors in the physical components and control the system using feedback loops in which physical processes influence computations and computations influence physical processes. Cyber-physical systems are present in many large industries such as electronics, energy, automotive, defense, aerospace, telecommunications, instrumentation and industrial automation.

An example of a contract in a cyber-physical system is a contract describing the preconditions and postconditions of an analog-to-digital converter (ADC) component. A natural language specification of such a contract is:

> *The ADC should output a digital voltage that is equal to the analog input voltage, within an error bound depending on the resolution of the ADC and the accuracy of the floating-point representation.*

According to what is written in [5], the main reason for introducing contract-based design is to mitigate delays in development and vulnerabilities in large, complex systems. The paper formalizes the notion of contracts in the context of system level design and presents a design methodology that allows for merging contracts with platform-based design. Platform-based design utilizes abstraction layers that describe functionality and architecture to achieve horizontal and vertical decompositions. Both horizontal and vertical integration is facilitated by using contracts. Contracts are described as “formalizations of the conditions for correctness of element integration (horizontal contracts),

for lower level of abstraction to be consistent with the higher ones, and for abstractions of available components to be faithful representations of the actual parts (vertical contracts)".

2.2 Meta-theory of contracts

The authors of the monograph [1] propose a theory of contracts theories, a mathematical meta-theory of contracts that focuses on system-level contracts. This entails that it gives a complete set of definitions on a high level of abstraction that can be concreted into concrete theories of contracts that suit certain contexts. Some of the definitions are considered primitive, and other are derived from those primitive ones. The important definitions of the meta-theory are summarized in this section, not including the proofs of the various concepts, theorems, and operations.

The meta-theory in [1] starts with the definition of *components*, while not specifying how to model them. A partially-defined associative and commutative *composition* operation is defined over pairs of components, $M_1 \times M_2$, and if the composition of two particular components M_1 and M_2 is defined, then the components is considered *composable*. Then, an *environment* for a component M is defined as a component E that is composable with M, meaning $E \times M$ is defined.

Next in [1], *contracts* are defined as pairs of sets of components. The *refinement* operation $C_1 \preceq C_2$ (read as C_1 refines C_2) is defined to mean that any implementation of C_1 is also a valid implementation of C_2. The *conjunction* operation $C_1 \wedge C_2$ between contracts is defined. The intuition behind contract conjunction is the combination of two contracts describing a single component. Defined also, is *contract composition* $C_1 \otimes C_2$, as being a greatest lower bound of C_1 and C_2. Intuitively, contract composition is the combination of two contracts that describe two different components into a single contract that describes both of the components together. Every definition is highly abstract, and the monograph contains complete proofs of all the derivations and theorems that are defined.

In [1], it is written that the meta-theory is to serve as a theory about contracts, from which more concrete theories shall be instantiated. Creating an instance of the meta-theory means to provide concrete explanations of the abstract definitions in the meta-theory. When many theories use the meta-theory as their basis, their concepts and definitions will be much more transferable since they are unified by the meta-theory. The monograph goes on to describe multiple instantiations of the meta-theory, the first one being a theory of so

called Assume/Guarantee contracts.

2.3 Assume/Guarantee contracts

The monograph [1] contains a definition of Assume/Guarantee contracts, which says that contracts are made up of assumptions and guarantees. Assumptions are analogous to the environments defined in the meta-theory, and guarantees are "the commitments of the component itself, when put in interaction with a valid environment". Here, the connection with the software contracts from [3] is clear, with the preconditions being assumptions and the postconditions being guarantees of the software component. Note that [3] is not intentionally an instantiation of the meta-theory in [1]. Indeed, the contracts for software development were developed very early, well before the formulation of the meta-theory.

An Assume/Guarantee contract C in [1] with assumptions A and guarantees G is written as $C = (A, G)$, where assumptions and guarantees are sets of assertions, an assertion being a behavior, e.g. a relationship between variables in a certain alphabet of variables. The monograph further defines the legal environments for C as the set of all components E such that $E \subseteq A$, and a particular component M implementing C is defined by $A \times M \subseteq G$. Finally, it is proven in the monograph that the Assume/Guarantee contracts theory is a true instantiation of the meta-theory.

2.4 Proving compositionality

The paper [4] by Nyberg et al deals with compositional verification, describing it as the act of verifying the behavior of each component contained in a large-scale system, rather than trying to verify the entire system directly. Nyberg et al define the challenge in compositional verification as proving that a decomposition of the system specification into the set of component specifications is done correctly. This challenge is called proving *compositionality*. The main contribution of the paper is the presentation of a proof system for proving compositionality in the general case of a non-monotonic system.

2.4.1 Monotonicity and non-monotonicity

In [6], the concept of non-monotonic composition, with respect to implementation, in contracts theories is explored. Monotonic composition refers

to composition of components implementing a specification following from one of the individual components implementing said specification. In monotonic composition, if a component C implements a specification S then the composition $C \times C'$ implements S, where C' is any component.

Many general contracts theories embed monotonic composition of components, but in [6] it is argued that contracts theories should embed non-monotonic composition instead. The reason is that specifying properties that only hold locally for a component is inherently not possible to express in a contracts theory embedding monotonic composition, since all implementing properties are preserved under composition. In [6], an example is presented that shows that if a specification specifies that a subsystem should not send out some signal, a subsystem implementing that specification cannot be part of a system that sends out that same signal and still maintain the property of not sending out the signal. A contracts theory embedding non-monotonic composition needs to explicitly address when composition is monotonic and when it is non-monotonic, since there are many cases where monotonic composition is desired.

2.4.2 Specification syntax

In the development of the proof system in [4] the idea of Assume/Guarantee contracts is used, letting a contract act as a specification of a component in a cyber-physical system. The contracts theory that is defined and used in [4] can be seen as an instantiation of the meta-theory in [1], although it may not have been developed with that purpose in mind. It is based on components $\mathcal{C}$, component composition $\mathcal{C}_1 \times \mathcal{C}_2$, specifications $\mathcal{S}$ and the implementation relation $\mathcal{C} : \mathcal{S}$, so there are certainly similarities between the theory and the meta-theory. The contracts theory is also non-monotonic, as described in section 2.4.1.

Nyberg et al go on to make some important definitions in [4], namely *behavior*, *behavior set*, *double intersection of behavior sets* and *downward-closed behavior set*. These definitions are summarized in the list below:

- Behavior: *A behavior* B *is a, possibly empty, set of runs.*
- Behavior set: *A behavior set* $\mathcal{Q}$ *is a, possibly empty, set of behaviors.*
- Double intersection of behavior sets: *The double intersection of two behavior sets* $\mathcal{Q}_1$ *and* $\mathcal{Q}_2$*, denoted* $\mathcal{Q}_1 \Cap \mathcal{Q}_2$*, is the behavior set* $\mathcal{Q}_1 \Cap \mathcal{Q}_2 ::= \{\mathsf{B}_1 \cap \mathsf{B}_2 | \mathsf{B}_1 \in \mathcal{Q}_1, \mathsf{B}_2 \in \mathcal{Q}_2\}$.

- Downward-closed behavior set: *A behavior set $\mathcal{Q}$ is downward closed if for each behavior* $\mathsf{B} \in \mathcal{Q}$*, it holds that each subset* $B' \subseteq B$ *is also in* $\mathcal{Q}$*, i.e.* $B' \in \mathcal{Q}$.

In the definitions above, *runs* are mentioned as the constituents of behaviors. In [4] Nyberg et al define runs as vectors of values of variables of a certain alphabet Ξ. The behavior made up of all the runs over Ξ is denoted by Ω.

There is much importance put on keeping the syntax and the semantics of the proof system apart. The syntactic elements in [4] are *component terms*, *specification terms*, *implementation judgments* and *specification judgments*. They are defined by the grammars listed below:

- Component term: $c ::= q \mid c \times c$
- Specification term: $s ::= p \mid s \sqcap s \mid (s, s) \mid s \parallel s$
- Implementation judgments: $\gamma ::= c : s$
- Specification judgments: $\phi ::= s \sqsubseteq s \mid \text{Assertional}(s)$

Where q is defined as an atomic component constant, s as an atomic specification constant, $c \times c$ as a composite component, $s \sqcap s$ as a conjunction of specifications, (s, s) as an Assume/Guarantee contract (as in section 2.3), $s \parallel s$ as a parallel composition and $s \sqsubseteq s$ as a refinement. $c : s$ should be read "c implements s". Beyond the definitions above, a specification ® is defined as the *implementability-ensuring specification* and $\top_{\parallel}$ is the *top specification*. The semantics of the concepts, including Assertional(s), is provided in [4] and is summarized below.

2.4.3 Specification semantics

The semantics of the syntax summarized above is a contribution from [4]. The first two important parts of the semantics are *interpretation* and *valuation*. The interpretation of a component or specification term in a model $\mathcal{M}$ is denoted $\langle . \rangle_{\mathcal{M}}$, where a model is a formalization of an engineering context. Interpretations map from terms to behaviors and behavior sets. Given this, the following interpretations are provided in [4]:

- $\langle q \rangle_{\mathcal{M}} \in 2^{\Omega}$
- $\langle p \rangle_{\mathcal{M}} \subseteq 2^{\Omega}$

- $\langle c_1 \times c_2 \rangle_{\mathcal{M}} = \langle c_1 \rangle_{\mathcal{M}} \cap \langle c_2 \rangle_{\mathcal{M}}$
- $\langle s_1 \sqcap s_2 \rangle_{\mathcal{M}} = \langle s_1 \rangle_{\mathcal{M}} \cap \langle s_2 \rangle_{\mathcal{M}}$
- $\langle s_1 \parallel s_2 \rangle_{\mathcal{M}} = \langle s_1 \rangle_{\mathcal{M}} \Cap \langle s_2 \rangle_{\mathcal{M}}$
- $\langle (a, g) \rangle_{\mathcal{M}} = \{\mathsf{B} \in 2^{\Omega} \mid \forall \mathsf{B}' \in \langle a \rangle_{\mathcal{M}}.\mathsf{B} \cap \mathsf{B}' \in \langle g \rangle_{\mathcal{M}}\}$
- $\langle \top_{\parallel} \rangle_{\mathcal{M}} = \{\Omega\}$
- $\langle ® \rangle_{\mathcal{M}} = \{\mathsf{B} \in 2^{\Omega} \mid \mathsf{B} \neq \varnothing\}$

Valuations are mappings from judgments to truth values, essentially evaluating a judgment. The three following judgments are provided in [4]:

- $[\![c : s]\!]_{\mathcal{M}} = T$ iff $\langle c \rangle_{\mathcal{M}} \in \langle s \rangle_{\mathcal{M}}$
- $[\![s_1 \sqsubseteq s_2]\!]_{\mathcal{M}} = T$ iff $\langle s_1 \rangle_{\mathcal{M}} \subseteq \langle s_2 \rangle_{\mathcal{M}}$
- $[\![\text{Assertional}(s)]\!]_{\mathcal{M}} = T$ iff $\langle s \rangle_{\mathcal{M}}$ is downward closed

In [4] it is noted that any specification $\mathcal{S}$ can be written as a contract $(\top_{\parallel}, \mathcal{S})$, and often contracts are used as the most general form of specification. The use for ® is to ensure implementability, and letting specifications be in conjunction with ®, i.e. $\mathcal{S} \sqcap ®$, is how Nyberg et al suggest implementability should be ensured.

The semantics of Assertional(s) is provided above, and the paper [4] also contains proposals of some properties about assertional specifications. An assertional specification $\mathcal{S}$ is monotonic with respect to composition, meaning that $\mathcal{C}_1 : \mathcal{S} \rightarrow \mathcal{C}_1 \times \mathcal{C}_2 : \mathcal{S}$ for any component $\mathcal{C}_2$. Nyberg et al also provide the intuition that an assertional specification can in general be expressed as a relation between variables, e.g. "$x < y$", and gives the following proposition about assertional contracts: *In a given model $\mathcal{M}$, a contract $(\mathcal{A}, \mathcal{G})$ is assertional if $\mathcal{G}$ is assertional or if $\langle \mathcal{A} \rangle_{\mathcal{M}} = \{\varnothing\}$.*

2.4.4 Compositionality conditions

Nyberg et al present a methodology in [4] for proving compositionality, where compositionality is defined as: *In a given engineering context represented by a model $\mathcal{M}$, a set of specifications $\mathcal{S}_1, ..., \mathcal{S}_N, N \geqslant 1$, are composable into a specification $\mathcal{S}$, or equivalently, the specification S is decomposable into the specifications $\mathcal{S}_1, ...\mathcal{S}_N, N \geqslant 1$, if*

$$[\![\mathcal{S}_1 \parallel ... \parallel \mathcal{S}_N \sqsubseteq \mathcal{S}]\!]_{\mathcal{M}} = T \tag{2.1}$$

The specifications are all assumed to be contracts (which, as previously mentioned, can be done without loss of generality), which yields

$$[\![(\mathcal{A}_1, \mathcal{G}_1) \parallel ... \parallel (\mathcal{A}_N, \mathcal{G}_N) \sqsubseteq (\mathcal{A}, \mathcal{G})]\!]_{\mathcal{M}} = T \tag{2.2}$$

To be able to prove equation 2.2 without any explicit reference to $\mathcal{M}$, Γ is introduced as a set containing additional specification judgements that hold true specifically in $\mathcal{M}$. Next, arbitrary components $q_0, ..., q_N$ implementing contracts $(\mathcal{A}_1, \mathcal{G}_1), ..., (\mathcal{A}_N, \mathcal{G}_N)$ respectively, are introduced. Then an attempt to find a formal proof of the sequent

$$\Gamma, q_1 : (\mathcal{A}_1, \mathcal{G}_1), ..., q_N : (\mathcal{A}_N, \mathcal{G}_N) \vdash q_1 \times ... \times q_N : (\mathcal{A}, \mathcal{G}) \tag{2.3}$$

can be made. The proof system used is a sound proof system of inference rules in the style of natural deduction.

2.4.5 Contract structures

Nyberg et al make use of a type of graph called a *contract structure* in [4], that contains all contracts considered (including their assumptions and guarantees) and refines relations between specification terms. A contract is represented by a link from an assumption to a guarantee and refinement is represented by links to assumptions and links from guarantees. The contract structure contains and visualizes the compositionality proof problem, except for the judgments Assertional(s). A contract structure provides intuition of what to prove and how to prove it, and can in certain special cases be sufficient by itself as a proof, see section 2.4.6.

[4] defines a contract structure as a graph adhering to the following rules. Given an engineering context represented by the model $\mathcal{M}$, a contract structure for a contract (A, G) and a set of sub-contracts $(A_i, G_i)_i$ is a Directed Acyclic Graph (DAG) such that

1. each node is an assumption A or A_i , or a guarantee G or G_i ,
2. each assumption has an outgoing arc to exactly one guarantee (these arcs define the contracts),
3. each incoming arc to each assumption A_i comes from a guarantee G_j where $i \neq j$, or from the assumption A,
4. the guarantee G has no outgoing arcs,
5. the assumption A has no incoming arcs,
6. each assumption A_i has at least one incoming arc,

7. there is at least one incoming arc to the guarantee G, and each such arc points from a guarantee G_i,
8. the guarantee G is in the model $\mathcal{M}$ refined by the conjunction of guarantees G_i with an outgoing arc pointing to G,
9. each assumption A_i is in the model $\mathcal{M}$ refined by the conjunction of specifications with an outgoing arc pointing to A_i.

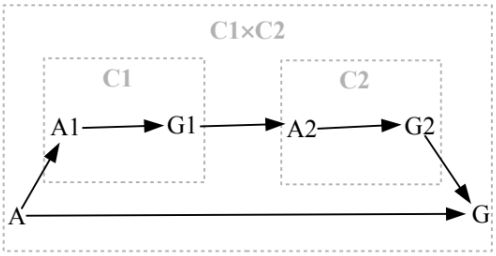

Figure 2.1: Contract structure from [4] with serial composition.

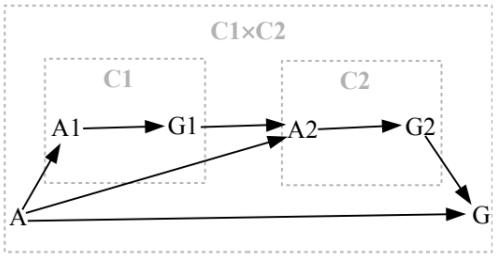

Figure 2.2: Contract structure from [4] in general non-monotonic case.

2.4.6 Special proof cases

Serial composition, according to Nyberg et al in [4], is applicable if in the contract structure each assumption $\mathcal{A}_i$ only has one incoming arc and also the guarantee $\mathcal{G}$ has an incoming arc from $\mathcal{A}$ and only from one $\mathcal{G}_i$. An example of a contract structure where serial composition is applicable can be seen in figure 2.1, and one where it is not applicable in figure 2.2. Serial composition is proven to be correct by incrementally composing individual arbitrary components implementing arbitrary contracts.

Another special case presented by Nyberg et al is proving compositionality in the monotonic case. It is proven in [4] that monotonic reasoning is possible

when all assumptions and guarantees are assertional. A theorem is presented that establishes that if all assumptions and guarantees in a contract structure are assertional, then equation 2.2 holds, proving compositionality.

2.4.7 Inference rules

For the general case, where serial composition is not applicable and all specifications cannot be assumed to be assertional, a proof system is presented in [4]. The proof system contains six inference rules shown in figure 2.3

Refinement $$\frac{c : s_1 \quad s_1 \sqsubseteq s_2}{c : s_2}\ \text{r}$$

Conjunction Introduction $$\frac{c : s_1 \quad c : s_2}{c : s_1 \sqcap s_2}\ \sqcap\text{i}$$

Conjunction Elimination $$\frac{c : s_1 \sqcap s_2}{c : s_1}\ \sqcap\text{e}$$

Contract Elimination $$\frac{c_1 : s_1 \quad c_2 : (s_1, s_2)}{c_1 \times c_2 : s_2}\ \text{ce}$$

Assertional Monotonicity $$\frac{c_1 : s \quad \text{Assertional}(s)}{c_1 \times c_2 : s}\ \text{am}$$

Contract Introduction $$\frac{\boxed{\begin{array}{l} q_0 \quad q_0 : s_1 \\ \qquad\ \vdots \\ \quad q_0 \times c : s_2 \end{array}}}{c : (s_1, s_2)}\ \text{ci}$$

Figure 2.3: The inference rules from [4].

There is also one derived rule presented, combining the rules conjunction elimination, refinement, and conjunction introduction. The rule is shown in figure 2.4.

Conjunction Refinement $$\frac{c : s_1 \sqcap s_2 \quad s_2 \sqsubseteq s_3}{c : s_2 \sqcap s_3}\ \text{cr}$$

Figure 2.4: The derived inference rule from [4].

The example in figure 2.2 is proven in [4] using the proof system. The proof, written as a proof DAG, can be seen in figure 2 as presented in [4].

Something to note is that the contracts theory and framework of [4] do not set any restrictions on how the specifications are expressed. This means that

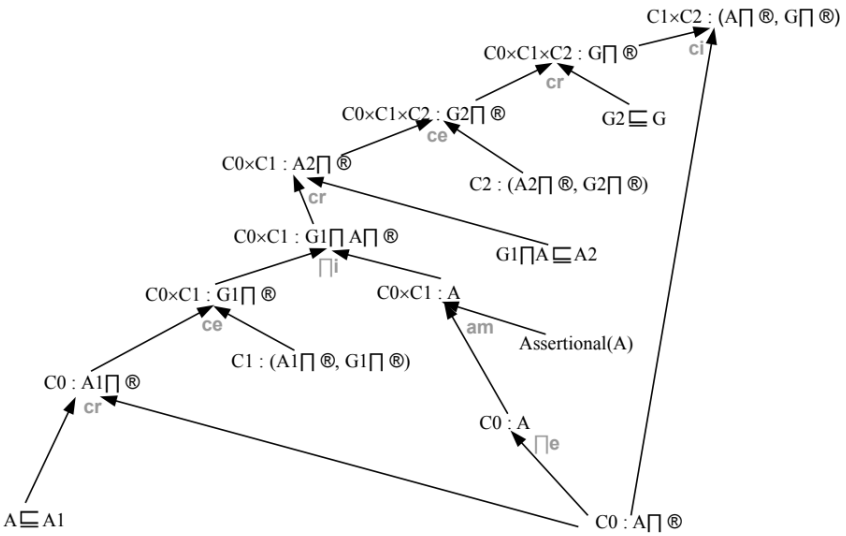

Figure 2.5: Proof DAG for a proof of the contract structure in figure 2.2, from [4].

the industrial applicability of the proof framework should be high. Constructing a contract structure and knowing whether the specifications are assertional or not is sufficient for using the framework.

Chapter 3

Method

In this chapter, the method of the study is laid forth. The approach is presented, as well as a motivation of the approach.

3.1 Overview

The utility of the framework in [4] was evaluated through applying the framework to a concrete example of a cyber-physical system. The example that was used is a Fuel Level Display (FLD) system presented in section 3.4. A top-level specification of the system was proposed in natural language, in accordance with what is described in [4]. The specification was decomposed following the system's architecture, producing a natural language specification for each component. The specifications were formalized into contracts described with formal logic definitions of assumptions and guarantees, using the concepts of [4]. Finally, the compositionality of the decompositions was proven using the framework contributed in[4], both using the special monotonic case of section 2.4.6 and using the inference rules of section 2.4.7 to carry out a formal proof.

3.2 Approach

Approaching the problem of creating decompositions required solving the modeling problem of defining a suitable language for writing specifications. Multiple languages were used, i.e. natural language requirement specifications were written and then formalized into a formal logic language.

A different modeling problem is to fix a time domain for the specification, which stems from defining how the time domains interrelate in the interfaces

between physical components and software components. Multiple time domains were used since the system in section 3.4 is a cyber-physical system with physical components as well as software components. Our approach to fixing multiple time domains was to fix a continuous time domain and adapt a discrete time domain with given discrete time steps within the continuous domain. A different, simpler approach was to disregard the execution times and delays in the software components, allowing the entire system to operate in a continuous time domain.

The following criteria were formulated for this study, and were considered in the process of evaluating the contracts theory:

- the degree of which the contracts theory supports formalizing natural language specifications into formal logic specifications,
- the degree of which the contracts theory supports fixing multiple time domains within a system,
- to what degree faithfulness of a specification in the contracts theory should be prioritized over simplicity, and
- the degree of which the proof framework scales well with the size of the system (number of components).

3.3 Motivation of approach

It is likely possible to imagine several evaluation criteria and evaluation methods that are not mentioned in section 3.2. However, in [4], the most prevalent modeling problems that are mentioned are the problems of specification language and time domain. Therefore, these are relevant criteria to measure in the evaluation. The criterion about formalizing natural language specifications is relevant because natural language specifications are common in real world engineering contexts according to [4]. Faithfulness and simplicity can in complex systems be considered opposites of each other, and considering which should be prioritized to what extent is therefore relevant. Proving compositionality in systems with a large amount of components could be a long process, and verifying that the proof framework does not give rise to unreasonably long compositionality proofs is therefore relevant.

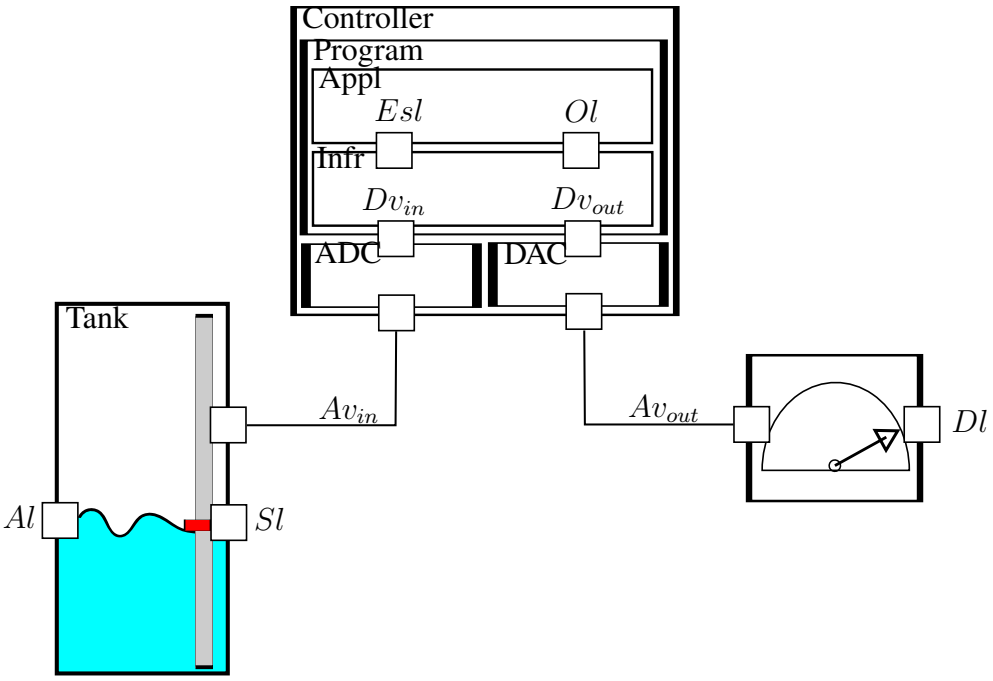

Figure 3.1: Example Fuel Level Display system. Variable explanations are available in section 3.4.

3.4 The FLD example

This section presents the FLD example acquired from [7]. The tasks from [7] serve as inspiration and motivation for the approach explained in 3.2.

The FLD system architecture shown in figure 3.1, consists of a Tank component, a Controller component and a Display component. The Tank component uses a slider connected to a floater to sense the actual fuel level in the tank. The Controller component contains the ADC, Program, and DAC subcomponents. The Program component in turn contains the two sub-components Appl (application software component) and Infr (infrastructure software component). The Controller takes a voltage from the Tank component as input and calculates an output voltage to the Display component, which displays an estimated fuel level corresponding to the output voltage.

The variables of the system are listed below.

- Al: Actual level [%]
- Sl: Sensed level [%] (Position of floater)
- Av_{in}: Analog voltage in [V] (W.r.t. ground)
- Dv_{in}: Digital voltage in [Float]

- Esl: Estimated sensed level [%]
- Ol: Output level [%]
- Dv_{out}: Digital voltage out [Float]
- Av_{out}: Analog voltage out [V] (W.r.t. ground)
- Dl: Displayed level [%]
- t: Time [s]

A set of tasks are also defined in [7], and are summarized below. These tasks are carried out and the results are presented in chapter 4.

1. Propose a top-level specification of the Fuel Level Display system. Intuitively, it should relate the displayed level Dl to the actual level Al over time. Note that one has to define a time domain, over which the specification is to be interpreted.

2. Come up with a decomposition along the architecture of the system. Note that there is a choice of specifying a component either directly with a single formula, or as an assume-guarantee pair of formulas.

3. Prove the correctness of the decomposition, using the proof rules in [4].

4. Argue for the realisability of the component specifications, in particular for specifications given as assume-guarantee pairs.

Chapter 4

Results

The results of this study consist of the decomposition of the top-level system specification of the Fuel Level Display (FLD) system, as well as the compositionality proof for the decomposition. The decomposition is presented in section 4.1 as natural language and formal specifications of each component in the decomposition. The proof is carried out in section 4.2 in two ways, first using the special monotonic case from section 2.4.6 and second using the inference rules from section 2.4.7 for a formal proof.

4.1 System specification decomposition

4.1.1 Decomposition

The decomposition along the architecture of the FLD system from 3.4 is shown in figure 4.1. Decomposition is done in multiple steps in a tree structure, until a leaf node is reached. Note that the decomposition does not strictly follow the architecture shown in figure 3.1, because of the introduction of the Sensor, Converter, InfrIn and InfrOut components. The specification for the component of the FLD node (root node) as well as the components of the leaf nodes are presented in section 4.1.2.

4.1.2 Component specifications

Natural language

The natural language specification for each leaf component as well as the top level specification from the decomposition shown in figure 4.1 are formulated as requirements and presented in table 4.1.

Table 4.1: Natural language specifications for all of the components.

Component	**Natural language specification (requirements)**
FLD	Should visually display a fuel level corresponding to the fuel level in the fuel tank, updated at given intervals in time.
Sensor	Should output a sensed level that is equal to the fuel level in the tank.
Converter	Should output a voltage proportional to the sensed level.
	The output voltage should lie between 0 V and a system-specific maximum output voltage.
ADC	Should output a digital voltage that is equal to the analog input voltage, within an error bound depending on the resolution of the ADC and the accuracy of the floating-point representation.
InfrIn	Should read a digital voltage at given intervals in time.
	Should output an estimated sensed fuel level proportional to the input voltage, within a fixed amount of time after reading the input.
Appl	Should read an estimated sensed fuel level at given intervals in time.
	Should output an output fuel level within a fixed amount of time after reading the input.
InfrOut	Should read an output fuel level at given intervals in time.
	Should output a digital voltage proportional to the output fuel level, within a fixed amount of time after reading the input.
DAC	Should output an analog voltage that is equal to the digital output voltage.
Display	Should visually display a fuel level, proportional to the analog output voltage.

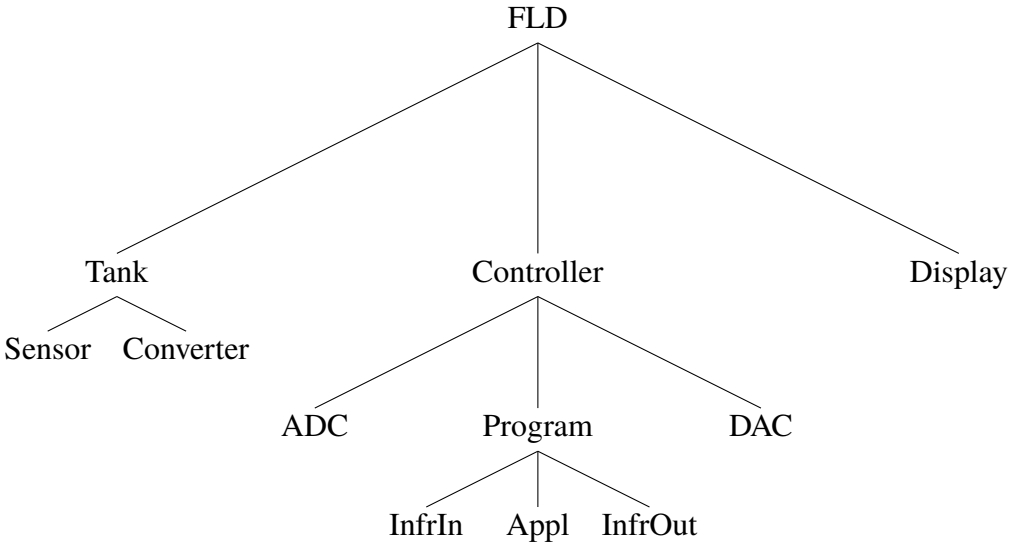

Figure 4.1: A decomposition tree from the top-level specification of the FLD system to the leaf nodes, which represent actual components.

Formal

The formal specifications are written as contracts, defining the input and output variables, the types of the variables, as well as the assumptions and guarantees. The variables in the system seen in figure 3.1 are listed in section 3.4. In addition to the given variables, time is added as the variable t, since all of the other variables are considered functions of time.

The specifications are parameterized by a set of constants defined as follows.

- Av_{in_max} is the maximum voltage that is possible to input into the Controller component, determined by the output voltage from the Tank component.
- Av_{out_max} is the maximum output voltage from the Controller component, determined by the input voltage to the display component.
- $T_{Program}$ is the time it takes for the Program component to run once, T_{Infr} is the time it takes for either of the InfrIn and InfrOut components to run once, and T_{Appl} is the time it takes for the Appl component to run once.
- P is the period of time that passes between each time the Program component reads the input voltage.

The following assumptions are also made about the parameters.

$Av_{in\_max}, Av_{out\_max}, T_{Program}, T_{Infr}, T_{Appl} > 0$ is necessary for physical correctness, $T_{Program} \in [0, P]$ specifies the requirement that the program executes faster than the period, and $2 \cdot T_{Infr} + T_{Appl} \leqslant T_{Program}$ defines the relationship between the execution times of the different software components that are part of the Program component.

For brevity, the specifications make use of a set of helper functions, defined in equations 4.1 and 4.2. The $\text{last}(t, \sigma)$ function in equation 4.2 calculates the greatest multiple of P that is lesser than t when summed with the offset σ. It is effectively used to get the point in time before t when a component with execution time σ finished its most recent execution, meant to show the delay that is required between software components.

$$\text{round}(x) = \text{rounds real number } x \text{ to floating point representation} \tag{4.1}$$

$$\text{last}(t, \sigma) = \max(\{m \in \{0P, 1P, 2P, ...\} \mid m + \sigma \leqslant t\}) \tag{4.2}$$

For each component, as well as for the top level specification, first the natural language specification is restated, then the formal specification is stated in the form of a contract. After each contract is a brief explanation and argument for implementability for the component.

The FLD system should visually display a fuel level corresponding to the fuel level in the fuel tank, updated at given intervals in time.

$$\mathcal{C}_{FLD} : \begin{cases} \text{variables} : \begin{cases} \text{inputs} : Al(t), t \\ \text{outputs} : Dl(t) \end{cases} \\ \text{types} : t \in \mathbb{R}, Al, Dl : \mathbb{R} \to \mathbb{R} \\ \text{assumption } A : Al(t) \in [0, 1], t > 0 \\ \text{guarantee } G : Dl(t) = \text{round}(Al(\text{last}(t, T_{Program}))) \end{cases}$$

The FLD contract is the top level specification. Al and Dl are mappings from time to percentage, measured as a real number between 0 and 1. Dl is guaranteed to be equal to Al at the last point in time Al was read, accounting for the program execution time, rounded to account for the floating point precision in the system.

The Sensor component should output a sensed level that is equal to the fuel level in the tank.

$$\mathcal{C}_{Sensor} : \begin{cases} \text{variables} : \begin{cases} \text{inputs} : Al(t), t \\ \text{outputs} : Sl(t) \end{cases} \\ \text{types} : t \in \mathbb{R}, Al, Sl : \mathbb{R} \to \mathbb{R} \\ \text{assumption } A_1 : Al(t) \in [0,1], t > 0 \\ \text{guarantee } G_1 : Sl(t) = Al(t) \end{cases}$$

The Sensor contract specifies the behavior of the Sensor component. Al and Sl are mappings from time to percentage, measured as a real number between 0 and 1. The Sensor component is assumed to work instantaneously, since its delay depends only on the speed of light.

An implementation of this component would not correspond exactly to the specification, since the specification guarantees equality between the sensed level and the actual level. In reality, there would be some small error in the sensor that the specification does not account for.

The Converter component should output a voltage proportional to the sensed level. The output voltage should lie between 0 V and a system-specific maximum output voltage.

$$\mathcal{C}_{Converter} : \begin{cases} \text{variables} : \begin{cases} \text{inputs} : Sl(t), t \\ \text{outputs} : Av_{in}(t) \end{cases} \\ \text{types} : t \in \mathbb{R}, Sl, Av_{in} : \mathbb{R} \to \mathbb{R} \\ \text{assumption } A_2 : Sl(t) \in [0,1], t > 0 \\ \text{guarantee } G_2 : Av_{in}(t) = Sl(t) \cdot Av_{in\_max} \end{cases}$$

The Converter contract specifies the behavior of the Converter component. Sl is a mapping from time to percentage, measured as a real number between 0 and 1. Av_{in} is a mapping from time to analog voltage. It is guaranteed that the level Sl maps linearly to Av_{in}. The Converter component is assumed to work instantaneously, since its delay depends only on the speed of light.

The implementability of the Converter component is dependent on the fact that the specification considers the relationship between the sensed level and the voltage to be linear. It is not unreasonable to assume that such a converter would be available. In reality, the sensor and converter components would likely be available as a single, off-the-shelf component for measuring fuel level in a tank and outputting a voltage. The case that the lowest output, indicating an empty tank, would not be 0 V in a real world component is easily adjusted for with a constant term added to the guarantee.

The ADC component should output a digital voltage that is equal to the analog input voltage, within an error bound depending on the resolution of the ADC and the accuracy of the floating-point representation.

$$\mathcal{C}_{ADC} : \begin{cases} \text{variables} : \begin{cases} \text{inputs} : Av_{in}(t), t \\ \text{outputs} : Dv_{in}(t) \end{cases} \\ \text{types} : t \in \mathbb{R}, Av_{in} : \mathbb{R} \to \mathbb{R}, Dv_{in} : \mathbb{R} \to \mathbb{Z} \\ \text{assumption } A_3 : Av_{in}(t) \in [0, Av_{in\_max}], t > 0 \\ \text{guarantee } G_3 : Dv_{in}(t) = \text{round}(Av_{in}(t)) \end{cases}$$

The ADC contract specifies the behavior of the ADC component. Av_{in} is a mapping from time to analog voltage, and Dv_{in} is a mapping from time to digital voltage. A digital voltage is represented as a floating point number stored as a sequence of bits, hence it is of type integer. The rounding of the analog voltage depends on the resolution of the ADC, where a higher resolution gives better precision. The sampling frequency of the ADC is not accounted for because it is significantly higher than the frequency P^{-1} (the sampling frequency of the Program component), so the ADC is assumed to work instantaneously.

The specification captures the behavior of a real analog-to-digital converter component adequately, the only omission is the sampling rate, which is considered irrelevant to the FLD system. Therefore, an off-the-shelf ADC would be able to fulfill the guarantee in the specification. The resolution of the ADC however, would affect the rounding error abstracted by round, and therefore the accuracy of the system.

The InfrIn component should read a digital voltage at given intervals in time and should output an estimated sensed fuel level proportional to the input voltage, within a fixed amount of time after reading the input.

$$\mathcal{C}_{InfrIn} : \begin{cases} \text{variables} : \begin{cases} \text{inputs} : Dv_{in}(t), t \\ \text{outputs} : Esl(t) \end{cases} \\ \text{types} : t \in \mathbb{R}, Dv_{in}, Esl : \mathbb{R} \to \mathbb{Z} \\ \text{assumption } A_4 : Dv_{in}(t) \in [0, \text{round}(Av_{in\_max})], t > 0 \\ \text{guarantee } G_4 : Esl(t) = \dfrac{Dv_{in}(\text{last}(t, T_{Infr}))}{\text{round}(Av_{in\_max})} \end{cases}$$

The InfrIn contract specifies the behavior of the InfrIn component. Dv_{in} is a mapping from time to digital voltage. A digital voltage is represented as a

floating point number stored as a sequence of bits, hence it is of type integer. Esl is a mapping from time to percentage, measured as a floating point number between 0 and 1. Dv_{in} is guaranteed to map linearly to Esl at the last point in time Dv_{in} was read, accounting for the InfrIn execution time, rounded to a floating point number.

The InfrIn component would not be available as an off-the-shelf component since it is too specific, and would have to be implemented specifically for the FLD system. A software program that reads a digital voltage as input and writes a variable as output to be input into another program is not an uncommon construct, and it is therefore realistic to assume that it would be implementable. Proving the correctness of this kind of program could be done easily using Hoare logic as in [2].

The Appl component should read an estimated sensed fuel level at given intervals in time and should output an output fuel level within a fixed amount of time after reading the input.

$$\mathcal{C}_{Appl} : \begin{cases} \text{variables} : \begin{cases} \text{inputs} : Esl(t), t \\ \text{outputs} : Ol(t) \end{cases} \\ \text{types} : t \in \mathbb{R}, Esl, Ol : \mathbb{R} \to \mathbb{Z} \\ \text{assumption } A_5 : Esl(t) \in [0,1], t > 0 \\ \text{guarantee } G_5 : Ol(t) = Esl(\text{last}(t, T_{Infr} + T_{Appl})) \end{cases}$$

The Appl contract specifies the behavior of the application software component. Esl is a mapping from time to percentage, measured as a floating point number between 0 and 1. Ol is a mapping from time to percentage, measured as a floating point number between 0 and 1. Floating point numbers are stored as sequences of bits, hence they are of type integer. Ol is guaranteed to be equal to Esl at the last point in time Esl was read, accounting for the Appl execution time.

Implementing the Appl component would simply be a matter of writing a program that waits for a new value of the input (or reads it at a specific interval determined by the system) and then copies it to the output. Proving the correctness of this kind of program could also be done easily using Hoare logic as in [2].

The InfrOut component should read an output fuel level at given intervals in time and should output a digital voltage proportional to the output fuel level, within a fixed amount of time after reading the input.

$$\mathcal{C}_{InfrOut} : \begin{cases} \text{variables} : \begin{cases} \text{inputs} : Ol(t), t \\ \text{outputs} : Dv_{out}(t) \end{cases} \\ \text{types} : t \in \mathbb{R}, Ol, Dv_{out} : \mathbb{R} \to \mathbb{Z} \\ \text{assumption } A_6 : Ol(t) \in [0,1], t > 0 \\ \text{guarantee } G_6 : Dv_{out}(t) = Ol(\text{last}(t, 2 \cdot T_{Infr} + T_{Appl}) \cdot \text{round}(Av_{out\_max}) \end{cases}$$

The InfrOut contract specifies the behavior of the InfrOut component. Ol is a mapping from time to percentage, measured as a real number between 0 and 1. Dv_{out} is a mapping from time to digital voltage. A digital voltage is represented as a floating point number stored as a sequence of bits, hence it is of type integer. Dv_{out} is guaranteed to be mapped linearly from Ol at the last point in time Ol was read, accounting for the InfrOut execution time, rounded to a floating point number.

The InfrOut component is very similar to the InfrIn component, and would in fact likely be part of the same software program. A software program that reads an input variable and writes a digital voltage as output is not an uncommon construct either, and it is therefore realistic to assume that it would be implementable. Proving the correctness of this kind of program could be done easily using Hoare logic as in [2], similar to the InfrIn component.

The DAC component should output an analog voltage that is equal to the digital output voltage.

$$\mathcal{C}_{DAC} : \begin{cases} \text{variables} : \begin{cases} \text{inputs} : Dv_{out}(t), t \\ \text{outputs} : Av_{out}(t) \end{cases} \\ \text{types} : t \in \mathbb{R}, Dv_{out} : \mathbb{R} \to \mathbb{Z}, Av_{out} : \mathbb{R} \to \mathbb{R} \\ \text{assumption } A_7 : Dv_{out}(t) \in [0, \text{round}(Av_{out\_max})], t > 0 \\ \text{guarantee } G_7 : Av_{out}(t) = Dv_{out}(t) \end{cases}$$

The DAC contract specifies the behavior of the DAC component. Dv_{out} is a mapping from time to digital voltage, and Av_{out} is a mapping from time to analog voltage. A digital voltage is represented as a floating point number stored as a sequence of bits, hence it is of type integer. The digital voltage is converted directly, without any rounding. The sampling frequency of the DAC is not accounted for because it is significantly higher than the frequency P^{-1}, so the DAC is assumed to work instantaneously.

Similar to the ADC specification, the DAC specification captures the behavior of a common off-the-shelf digital-to-analog converter component. In this case, the guarantee does not take rounding error into consideration, since the signal degradation of the DAC is assumed to be insignificant. The sampling frequency is also ignored, with the same reasoning as for the ADC component.

The Display component should visually display a fuel level, proportional to the analog output voltage.

$$\mathcal{C}_{Display} : \begin{cases} \text{variables} : \begin{cases} \text{inputs} : Av_{out}(t), t \\ \text{outputs} : Dl(t) \end{cases} \\ \text{types} : t \in \mathbb{R}, Av_{out}, Dl : \mathbb{R} \rightarrow \mathbb{R} \\ \text{assumption } A_8 : Av_{out}(t) \in [0, Av_{out\_max}], t > 0 \\ \text{guarantee } G_8 : Dl(t) = \dfrac{Av_{out}(t)}{Av_{out\_max}} \end{cases}$$

The Display contract specifies the behavior of the Display component. Av_{out} is a mapping from time to analog voltage. Dl is a mapping from time to percentage, measured as a real number between 0 and 1. It is guaranteed that the voltage Av_{out} maps linearly to Dl. The Display component is assumed to work instantaneously, since its delay depends only on the speed of light.

The specification does not specify exactly what type of real display that would be used. This is kept abstract since that level of detail is not considered necessary. An off-the-shelf analog or digital display that is commonly found in vehicle dashboards would be adequate for fulfilling the guarantee of the specification. A consequence of using an analog display is that a certain constant delay is added to the last step of the FLD system. This is not modeled in the specification since the type of display is kept abstract.

4.1.3 No delay case

The decomposition specified in section 4.1.2 takes delays into account. Specifically, the InfrIn, Appl and InfrOut components have delays. A different specification that can be made takes no delays into account, assuming the FLD system to work instantaneously. In this case, the specifications would look different. All calls to the function $\mathrm{last(t}, \sigma)$ would be replaced by simply using the variable t.

In reality, implementing the system according to a specification without delays would likely prove difficult. It is necessary for the software components

in the system to have some delay since the computation in one component is dependent on previous components' computations, and the computations are not instantaneous in reality.

4.2 Compositionality proof

Proving the compositionality of the decomposition of the FLD system involves three parts: proving the specifications to be assertional, creating a contract structure and carrying out a natural deduction-style proof using the inference rules in [4]. In the monotonic case, a contract structure alone is enough to prove the compositionality of the decomposition, according to what is explained in section 2.4.6. However, to further the goal of evaluating the compositionality proof framework, the compositionality will also be proven using the inference rules from section 2.4.7.

Proving the compositionality of the decomposition amounts to proving the sequent 4.3, where Γ is defined in equation 4.4.

$$\begin{gathered}\Gamma, \text{Sensor} : (A_1 \sqcap \circledR, G_1 \sqcap \circledR), \text{Converter} : (A_2 \sqcap \circledR, G_2 \sqcap \circledR), \\ \text{ADC} : (A_3 \sqcap \circledR, G_3 \sqcap \circledR), \text{InfrIn} : (A_4 \sqcap \circledR, G_4 \sqcap \circledR), \text{Appl} : (A_5 \sqcap \circledR, G_5 \sqcap \circledR), \\ \text{InfrOut} : (A_6 \sqcap \circledR, G_6 \sqcap \circledR), \text{DAC} : (A_7 \sqcap \circledR, G_7 \sqcap \circledR), \text{Display} : (A_8 \sqcap \circledR, G_8 \sqcap \circledR) \\ \vdash \\ \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times \text{Appl} \times \text{InfrOut} \times \text{DAC} \times \text{Display} : \\ (A \sqcap \circledR, G \sqcap \circledR)\end{gathered} \tag{4.3}$$

$$\begin{gathered}\Gamma = A \sqsubseteq A_1, G_1 \sqcap A \sqsubseteq A_2, G_2 \sqcap A \sqsubseteq A_3, G_3 \sqcap A \sqsubseteq A_4, G_4 \sqcap A \sqsubseteq A_5, \\ G_5 \sqcap A \sqsubseteq A_6, G_6 \sqcap A \sqsubseteq A_7, G_7 \sqcap A \sqsubseteq A_8, G_8 \sqsubseteq G, \\ \text{Assertional}(A), \text{Assertional}(A_1), \text{Assertional}(A_2), \text{Assertional}(A_3), \\ \text{Assertional}(A_4), \text{Assertional}(A_5), \text{Assertional}(A_6), \text{Assertional}(A_7), \\ \text{Assertional}(A_8), \text{Assertional}(G_1), \text{Assertional}(G_2), \text{Assertional}(G_3), \\ \text{Assertional}(G_4), \text{Assertional}(G_5), \text{Assertional}(G_6), \text{Assertional}(G_7), \\ \text{Assertional}(G_8), \text{Assertional}(G)\end{gathered} \tag{4.4}$$

Assertionality

As explained in section 2.4.3, if all specifications are assertional, it means that the case is monotonic. The formal definition of assertionality is that an assertional specification describes a behavior set that is downward closed, as explained in section 2.4.3. An assertional specification is less formally but equivalently one that can be expressed as a simple relation between variables. Since all specifications are expressed in this way, they are assertional.

Contract structure

As previously established, the decomposition displays monotonicity, which as stated in section 2.4.3 implies that a contract structure is enough to prove compositionality. The contract structure of the present decomposition is shown in figure 4.2, and can be seen to correspond to sequent 4.3. The structure is also consistent with the rules that govern contract structures, laid out in section 2.4.5.

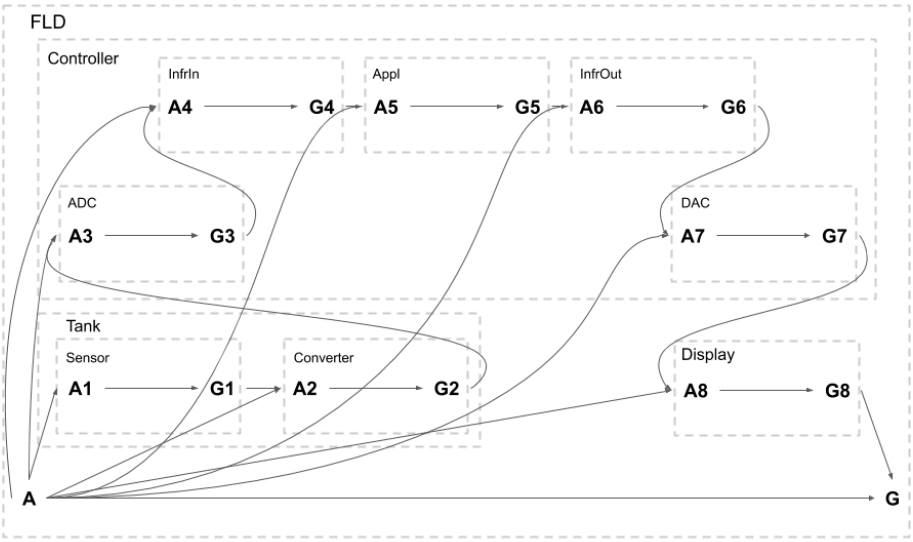

Figure 4.2: Contract structure for the FLD decomposition.

Formal proof

A proof of the compositionality of the decomposition is presented in table 4.2, and conclusively proves the compositionality. The inference rules from [4] described in section 2.4.7 are used in this section.

Table 4.2: Natural deduction-style proof of compositionality in the FLD specification decomposition.

1	$A \sqsubseteq A_1$	Premise
2	$G_1 \sqcap A \sqsubseteq A_2$	Premise
3	$G_2 \sqcap A \sqsubseteq A_3$	Premise
4	$G_3 \sqcap A \sqsubseteq A_4$	Premise
5	$G_4 \sqcap A \sqsubseteq A_5$	Premise
6	$G_5 \sqcap A \sqsubseteq A_6$	Premise
7	$G_6 \sqcap A \sqsubseteq A_7$	Premise
8	$G_7 \sqcap A \sqsubseteq A_8$	Premise
9	$G_8 \sqsubseteq G$	Premise
10	$\mathrm{Assertional}(A)$	Premise
11	$\mathrm{Sensor} : (A_1 \sqcap ®, G_1 \sqcap ®)$	Premise
12	$\mathrm{Converter} : (A_2 \sqcap ®, G_2 \sqcap ®)$	Premise
13	$\mathrm{ADC} : (A_3 \sqcap ®, G_3 \sqcap ®)$	Premise
14	$\mathrm{InfrIn} : (A_4 \sqcap ®, G_4 \sqcap ®)$	Premise
15	$\mathrm{Appl} : (A_5 \sqcap ®, G_5 \sqcap ®)$	Premise
16	$\mathrm{InfrOut} : (A_6 \sqcap ®, G_6 \sqcap ®)$	Premise
17	$\mathrm{DAC} : (A_7 \sqcap ®, G_7 \sqcap ®)$	Premise
18	$\mathrm{Display} : (A_8 \sqcap ®, G_8 \sqcap ®)$	Premise
19	$C_0.\ C_0 : A \sqcap ®$	Assumption
20	$C_0 : A_1 \sqcap ®$	cr 19, 1
21	$C_0 \times \mathrm{Sensor} : G_1 \sqcap ®$	ce 20, 11
22	$C_0 : A$	$\sqcap$e 19
23	$C_0 \times \mathrm{Sensor} : A$	am 22, 10
24	$C_0 \times \mathrm{Sensor} : G_1 \sqcap A \sqcap ®$	$\sqcap$i 21, 23
25	$C_0 \times \mathrm{Sensor} : A_2 \sqcap ®$	cr 24, 2
26	$C_0 \times \mathrm{Sensor} \times \mathrm{Converter} : G_2 \sqcap ®$	ce 25, 12
27	$C_0 \times \mathrm{Sensor} \times \mathrm{Converter} : A$	am 23, 10
28	$C_0 \times \mathrm{Sensor} \times \mathrm{Converter} : G_2 \sqcap A \sqcap ®$	$\sqcap$i 26, 27
29	$C_0 \times \mathrm{Sensor} \times \mathrm{Converter} : A_3 \sqcap ®$	cr 28, 3
30	$C_0 \times \mathrm{Sensor} \times \mathrm{Converter} \times \mathrm{ADC} : G_3 \sqcap ®$	ce 29, 13
31	$C_0 \times \mathrm{Sensor} \times \mathrm{Converter} \times \mathrm{ADC} : A$	am 27, 10
32	$C_0 \times \mathrm{Sensor} \times \mathrm{Converter} \times \mathrm{ADC} : G_3 \sqcap A \sqcap ®$	$\sqcap$i 30, 31
33	$C_0 \times \mathrm{Sensor} \times \mathrm{Converter} \times \mathrm{ADC} : A_4 \sqcap ®$	cr 32, 4
34	$C_0 \times \mathrm{Sensor} \times \mathrm{Converter} \times \mathrm{ADC} \times \mathrm{InfrIn} : G_4 \sqcap ®$	ce 33, 14
35	$C_0 \times \mathrm{Sensor} \times \mathrm{Converter} \times \mathrm{ADC} \times \mathrm{InfrIn} :$	

	A	am 31, 10
36	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} :$ $G_4 \sqcap A \sqcap ®$	$\sqcap$i 34, 35
37	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} :$ $A_5 \sqcap ®$	cr 36, 5
38	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} : G_5 \sqcap ®$	ce 37, 15
39	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} : A$	am 35, 10
40	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} : G_5 \sqcap A \sqcap ®$	$\sqcap$i 38, 39
41	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} : A_6 \sqcap ®$	cr 40, 6
42	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} \times \text{InfrOut} : G_6 \sqcap ®$	ce 41, 16
43	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} \times \text{InfrOut} : A$	am 39, 10
44	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} \times \text{InfrOut} : G_6 \sqcap A \sqcap ®$	$\sqcap$i 42, 43
45	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} \times \text{InfrOut} : A_7 \sqcap ®$	cr 44, 7
46	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} \times \text{InfrOut} \times \text{DAC} : G_7 \sqcap ®$	ce 45, 17
47	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} \times \text{InfrOut} \times \text{DAC} : A$	am 43, 10
48	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} \times \text{InfrOut} \times \text{DAC} : G_7 \sqcap A \sqcap ®$	$\sqcap$i 46, 47
49	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} \times \text{InfrOut} \times \text{DAC} : A_8 \sqcap ®$	cr 48, 8
50	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} \times \text{InfrOut} \times \text{DAC} \times \text{Display} : G_8 \sqcap ®$	ce 49, 18
51	$C_0 \times \text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times$ $\text{Appl} \times \text{InfrOut} \times \text{DAC} \times \text{Display} : G \sqcap ®$	cr 50, 9
52	$\text{Sensor} \times \text{Converter} \times \text{ADC} \times \text{InfrIn} \times \text{Appl} \times$ $\text{InfrOut} \times \text{DAC} \times \text{Display} : (A \sqcap ®, G \sqcap ®)$	ci 19-51

Chapter 5

Discussion

This chapter starts with the evaluation of the contracts theory, a major part of this report. After the evaluation, what can be learned from the results is discussed, as well as the project's limitations.

5.1 Evaluation

The main subject of discussion in this report is the evaluation of the contracts theory and proof framework. Criteria for the evaluation are presented in the itemized list in section 3.2, and will be repeated in the following sections. They are referred to by their position in the list in 3.2, e.g. *the first criterion*, *the second criterion*. Lastly, some drawbacks of the framework are brought up.

5.1.1 Formalization of natural language

The first criterion, *the degree of which the contracts theory supports formalizing natural language specifications into formal logic specifications*, is easily evaluated. After formulating specifications as requirements in natural language, the syntactic constructs and semantics defined in the contracts theory were useful for writing coherent logical expressions using common logical and mathematical symbols. Despite the fact that the contracts theory does not mandate how specifications should be written, common logical expressions are very well suited for this purpose. Allowing runs, as described in [4], to be defined by logical expressions simplifies the specification process significantly. Writing specifications as runs and sets of runs directly appears to be an unnecessarily imprecise and laborious approach. Letting logical expressions

define the runs seems to be a simple and sufficient approach for formalizing natural language specifications.

5.1.2 Faithfulness and complexity

The second and third criteria both relate to the question of faithfulness and complexity. The criteria are *the degree of which the contracts theory supports fixing multiple time domains within a system* and, *to what degree faithfulness of a specification in the contracts theory should be prioritized over simplicity*. Specifications with a high degree of faithfulness to the real world tend to be more complex, especially when fixing multiple time domains and modeling delays in a system. However, this is likely not to be unique for the contracts theory in [4], since the way specifications are defined is in line with the meta-theory from [1].

With the FLD system, it was easy to define a specification that assumes no delays in any component, as shown in section 4.1.3. This specification has relatively low faithfulness to the real world, compared to the specification in section 4.1.2. In that specification, the software components have delays, while the other components are assumed to work instantaneously. This is a reasonable level of faithfulness, since the software components are assumed to have delays several orders of magnitude larger than the potential delays in the rest of the system.

Fixing multiple time domains could likely be done in multiple different ways. In the specification of section 4.1.2, a continuous time domain is assumed, and a period is added to discretize the system according to the needs of the software components. This seems like an approach that stays faithful to real world behavior, and is well suited for use in the confines of the contracts theory. In order for the approach to work, however, the period and computation time parameters were added, along with other parameters. More complex systems where multiple separate discrete time domains need to be specified might require a large number of parameters, leading to a high complexity of the specification. In such cases, the approach used in section 4.1.2 may be impractical, and other approaches should be explored.

The discrete output values of real-life ADC and DAC components were abstracted in the specifications that were made, and replaced by a rounding error. This could reasonably be done since an off-the-shelf ADC or DAC component should have a high enough resolution for it to not matter in the context of the system. It also helped keeping the complexity of the specification down.

Determining what to prioritize in regards to faithfulness and complexity is

highly dependent on the context of the system. Modeling small details such as delays might be necessary in some cases, but quickly makes the system specifications more complex. In the case of the FLD system, the level of detail required in the specification was carefully considered and placed at an appropriate level for the context. However, other systems in other contexts might require that details such as the delay that the speed of light introduces be included. An advantage of the framework is that because of its high level of abstraction, it is adaptable to various contexts with different requirements on the level of detail.

5.1.3 Scaling

The fourth criterion, *the degree of which the proof framework scales well with the size of the system (number of components)*, concerns the scaling of the specifications in relation to system size. Essentially, as the number of different components in a system grows, the size of contract structures and length of proofs, as well as their clarity, should remain manageable.

The FLD system is a system with a relatively small amount of components. Despite this, the contract structure in figure 4.2 is both large and complex. The proof presented in table 4.2 is also long and could be difficult to follow. With these results, it is plausible to assume that the complexity of contract structures and proofs generally do not scale well with the size of the system.

A solution to this problem could be to make use of more intermediary steps and prove the correctness of sub-components containing a subset of components before proving the correctness of the whole decomposition. In the FLD system, proving compositionality in sub-components should follow the decomposition hierarchy shown in figure 4.1. For a system of a similar size to the FLD system, proving compositionality in sub-components might not be necessary since the system is still relatively small. In larger systems, using the framework to prove compositionality in sub-components could become a necessity.

In real applications, it would be unnecessary to find compositionality proofs manually. It is likely possible to design a technique for automating the proof search process for compositionality proofs that use the framework of [4]. An automated proof search would render the scaling issues of the formal proofs irrelevant, since no humans would have to inspect the proofs. However, the scaling issues of the contract structures would not be solved by an automated proof search, since their purpose is partially to visualize a decomposition to humans.

5.1.4 Miscellaneous drawbacks

During the construction of the proof, two minor drawbacks of the framework were discovered. Firstly, the inference rules do not reflect the inherent commutativity of the syntactic constructs in the framework. When using inference rules to prove a sequent, consistent syntax is important in order to ensure the correct application of the rules. In multiple positions in the proof in table 4.2, the order of operands has been modified between the lines in order to make a presentable proof that applies the inference rules in as close as possible to a satisfactory manner. It might improve the quality of the framework if additional rules are added, in order to reflect the commutativity of the framework's operators.

The second drawback concerns the implementability-ensuring specification. The symbol for the specification (®) always needs to be included in order to reflect real-world behavior, which is completely reasonable, but since it is always included it becomes somewhat redundant. Letting the implementability-ensuring specification be implicitly included might improve the readability of the proofs, and it also might remove some of the problems with commutativity mentioned as the first drawback above. The issue is not with the intent of the implementability ensuring-specification, but with the fact that it always needs to be written explicitly.

5.2 Project analysis

5.2.1 Contributions

The results of this study could have some implications for the contracts research field. In the case that further research related to the contracts theory of [4] is conducted, our results could be useful. For instance, the results could give insight into one possible way of applying the framework on a concrete system. The results could help in expediting the understanding of the original paper [4], which might improve the quality of the research in the end.

5.2.2 Limitations

A significant limitation of this study is that only one example of a top level system specification was decomposed, and its decomposition proven correct. Having several decompositions and proofs of decompositions would likely be beneficial for the evaluation of the contracts theory and framework. Fur-

thermore, the example that was used was a simple system that was inherently monotonic in nature. Since an important feature of the framework and contracts theory is the support for non-monotonicity, an example system with non-monotonic properties would have been better suited for the purpose of evaluating the theory and framework.

Chapter 6

Conclusion

In this study an evaluation of the contracts theory and proof framework proposed by Nyberg et al in [4] was carried out, by decomposing an example of a top-level specification and proving the compositionality of the decomposition using the framework.

As a conclusion, the contracts theory's syntax and semantics made it simple to formalize natural language specifications into descriptions of runs. The theory and framework are also well suited for adaptation to different contexts with different requirements, making the modeling gap between the theory's level of abstraction and the concrete system easy to close. A problem with the framework is that the length and complexity of the proofs created using it scale poorly with the size and complexity of the decomposition. A way to mitigate this problem could be to introduce more layers of abstraction and to prove the compositionality of smaller groups of components before proving the compositionality of the whole system. Automating the proof search would render the scaling problem irrelevant. Overall, the contracts theory and framework have high utility in most regards.

The evaluation in this report does not include the non-monotonic case. A possible subject for future work is to complete the evaluation by exploring the framework's utility in a system with some degree of non-monotonicity.

Acknowledgments

We would like to thank our supervisor in this project, Dilian Gurov, who has always been available when we needed guidance throughout this project. We would also like to extend our gratitude to the other two authors of the paper [4] that this report is based on, Mattias Nyberg and Jonas Westman. Additionally, we want to thank the two of our peers who peer reviewed the report, and our friends and family who took the time to proofread it.

Bibliography

[1] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. Larsen. "Contracts for System Design". eng. In: *Foundations and Trends in Electronic Design Automation* 12.2-3 (2018), pp. 124–400. ISSN: 1551-3939.

[2] C.A.R. Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[3] B. Meyer. "Applying 'design by contract'". eng. In: *Computer* 25.10 (1992), pp. 40–51. ISSN: 0018-9162.

[4] M. Nyberg, J. Westman, and D. Gurov. "Proving Compositionality in a Non-Monotonic Contracts-Theory". Unpublished Manuscript. 2019.

[5] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems". eng. In: *European Journal of Control* 18.3 (2012), pp. 217–238. ISSN: 0947-3580.

[6] M. Nyberg and J. Westman. "Preserving Contract Satisfiability Under Non-monotonic Composition". eng. In: vol. LNCS-10854. 2018, pp. 181–195. ISBN: 978-3-319-92611-7.

[7] M. Nyberg, J. Westman, D. Gurov, P. Filipovikj, and P. Lidström. "Fuel Level Display: An Exercise in Requirements Decomposition". Unpublished Manuscript. 2019.

TRITA -EECS-EX-2020:394