Formal Knowledge Representations for Textual Automotive System Requirements and Tests

A thesis accepted by the Faculty of Aerospace Engineering and Geodesy of the University of Stuttgart in partial fulfilment of the requirements for the degree of Doctor of Engineering Sciences (Dr.-Ing.)

by

M.Sc. Benedikt Walter

born in Ellwangen (Jagst)

main referee:	Priv.-Doz. Dr.-Ing. Stephan Rudolph
co-referee:	Prof. Dr. Andreas Vogelsang
co-referee:	Prof. Dr.-Ing. Andreas Strohmayer
Date of defence:	03.12.2020

Institute of Aircraft Design
University of Stuttgart
2021

"Whenever you find yourself on the side of the majority, it is time to pause and reflect."

Mark Twain

Preamble

This thesis is based on the research conducted during my time as an industrial PhD student at Daimler AG, Mercedes-Benz car development between June 2015 and April 2018. It was finalized during my time as a development engineer in the Department of Automated Driving from May 2018 until the end of 2020. The research took place in the E/E System Testing and Validation Department where I worked for 1.5 years in the Team *'Testing Process'* and 1.5 years in the Team *'Requirements Validation and Testing Methodology'*. In particular I want to thank Dirk Johanson (Lead of the Team *'Testing Process'*) for providing me with this opportunity and the freedom to pursue my research into any direction I was interested in. Further, I want to thank Dr. Frank Houdek (Lead of the Team *'Requirements Validation and Testing Methodology'*) for his support and helpful guidance in the art of writing scientific papers.

Many thanks go to all students that worked along side me throughout the time: Ivan Vishev, Jakob Hammes, Marco Piechotta, Melf Zeymer, Jonathan Schmidt and Hannah Dettki. This work would have not been possible without all your effort and dedication. In the same way I want to thank the young researchers that were brave enough to perform their bachelor or master thesis under my supervision: Mohammed Salah Ben Slimen, Maximilian Schilling, Jan Martin and Payal Sood. Your ideas, propositions and solutions to difficult problems were a big help in publishing the conducted research and putting this work together.

The research was supported in big parts by the University of Stuttgart, in particular by Priv-Doz. Dr. Stephan Rudolph at the Institute of Aircraft Design. I can not thank you enough for all the hours spent with white paper, pencil and coffee, sketching out and constructing this research and guiding me through emotional and intellectual high mountains and deep valleys during my time as a PhD student. Without your help this work would have not happened!

Further, my thanks goes to Prof. Dr. Andreas Vogelsang at the Berlin Institute of Technology for ideas, comments and critics that helped shaping this work. In addition, I want to thank Robin Loose from the Department of Mathematics at the University of Münster for his effort and help with the mathematical foundation of the work.

Last, I want to thank my family for their support, help and advice throughout the almost six years that this project consumed. Thanks to my parents Andrea & Ulrich Walter for everything they supported me with during this time. Thanks to my siblings Jonas & Lea Walter which were always curious and interested in what I did. Finally, I want to thank my wife Christina Colondres Walter for her patience, kindness and support whenever it was needed.

This said, I let the reader now explore this work and hope that it sparks some inspiration and discussion. It hopefully resolves some questions and might raise some additional ones, but this is what science is all about.

Benedikt Walter

Contents

List of Figures

List of Tables

Acronyms

AOLC Adaptive Outside Light Control

Atomic FSM Atomic Finite State Machine

Atomic Requirement FSM Atomic Requirement Finite State Machine

CAD Computer-Aided Design

CFD Computational Fluid Dynamics

C-HIL Compoenent-HIL

CNF Conjunctive Normal Form

CTL Computational Tree Logic

DBSCAN Density-Based Spatial Clustering

DC43 Design Cockpit 43

DNF Disjunctive Normal Form

E/E electric/electronic

FEM Fintite Element Analysis

FMU Functional Mock-Up

FOL First Order Logic

FSM Finite State Machine

GIL Graphical Interval Logic

GUI Graphical User Interface

HIL Hardware-in-the-Loop

ICSFSM Incomplete Specified Finite State Machine

IEEE Institute of Electrical and Electronics Engineers

ILS Intelligent Light System

ISO International Organization for Standardization

ISTQB International Software Testing Qualifications Board

K-Means Centroid Based

LHS Left-Hand Side

LTL Linear Temporal Logic

M2M Model-to-Model

M2T Model-to-Text

MBC Mercedes-Benz Passenger Cars

Mealy DFSM Mealy Deterministic FSM

Mealy FSM Mealy Finite State Machine

MIL Model-in-the-Loop

Moore DFSM Moore Deterministic FSM

Moore FSM Moore Finite State Machine

NFR Non-Functional Requirement

NL Natural Language

OEM Original Equipment Manufacturer

OLC Outside Light Control

PDP Product Development Process

PNDFSM Partially Non-Deterministic Finite State Machine

ReqIF Requirements Interchange Format

Requirement FSM Requirement Finite State Machine

RHS Right-Hand Side

SIL Software-in-the-Loop

SLINK Hierarchical Single-Linkage

SPS Specification Pattern Systems

STG State-Transition Graph

System FSM System Finite State Machine

T2M Text-to-Model

UML Unified Modeling Language

Abstract

The current control systems in the automotive domain exceed more and more often the size where engineers can manually perform certain tasks in the field of requirements engineering. Even for humans it is hardly possible to compare 2000 or more requirements in natural language form for correctness, consistency or completeness. One solution to this problem is to achieve a machine-readable representation form of the requirements in order to automate these tasks. This approach results in a knowledge transformation process that can be applied to automotive requirements and tests. With this process, requirements can be converted from textual representation into a machine-readable form. Such a representation shall here be called 'formalized' representation.

The most important result of this work is an approach which allows to process such formalized representations. The processing of formalized representations can be automated to perform decision making for system design decisions, analytical processing and reasoning about the system and code generation within in the system design process. The derived formalization process can be broken down into the following parts: The initial natural language representations of requirements and tests are at first converted manually (by an engineer) to logic expressions with the use of specification patterns. These patterns are re-used from preexisting work. Logic expressions allow sequential boolean transformations and all further transformations can therefore be proven to be analytically correct. The initial logic representation occurs in linear temporal logic, which is compact but not explicit for a particular state in a system. Transformation to first order logic and reordering in conjunctive normal form achieves a state-wise explicit normal form. A representation of requirements and tests in state-wise conjunctive normal form allows for reasoning about redundancy and consistency for a given set of requirements or tests. These state-wise represented requirements can be merged into one single state machine which represents the overall system. The derived system state machine is analytically correct and provides the basis for further analysis.

The consequences of the existence of such an analytical correct conversion process is, that the problems of correctness, consistency and completeness are closely connected to a representation problem. These problems may be solved whenever an initial semi-

formal, pattern-based representation can be established. The introduced processing chain is described with a step-wise conversion, illustrated with examples for requirements and tests. Correctness, consistency and completeness of requirements or tests for a system can be shown with a system representation in form of state machines. This work shows a solution to convert logic expressions into dynamic finite state machines. The correctness and applicability of this approach are illustrated with industrial case studies. The formalized representation form shown in the approach allows to addresses the problems of requirements with respect to correctness, consistency and transformation between equivalent representation forms. In addition, the effects of scalability through application to industrial systems are investigated and discussed.

Overall, this approach achieves a formalization process for requirements and tests. It shows the correctness, applicability and scalability of the process and its conversion steps in industrial contexts. Therefore this approach represents a solution approach to the problem of automated requirements processing. This approach represents one solution to integrate the domain of requirements engineering within a fully digital system design process.

Kurzfassung

Die Systeme im Automotivbereich überschreiten mehr und mehr eine Größe, bei der Ingenieure in der Lage sind, bestimmte Tätigkeiten im Requirements-Engineering manuell durchzuführen. Es ist nur schwer möglich oder übersteigt sogar die menschlichen Fähigkeiten 2000 oder mehr Anforderungen in natürlicher Sprache auf Korrektheit, Konsistenz und Vollständigkeit zu prüfen. Eine Lösung für dieses Problem ist ein Ansatz zur Formalisierung der Anforderungen, um diese Tätigkeiten zu automatisieren. Daher wird in dieser Arbeit ein Ansatz entwickelt, der zur Wissensformalisierung von Anforderungs- und Testdaten im Automotivbereich genutzt werden kann. Mit diesem Prozess können die textuellen Anforderungen in maschinenlesbare Form gebracht werden. Solch eine Darstellung soll hier als 'formalisiert' bezeichnet werden.

Das wichtigste Ergebnis dieser Arbeit ist ein Ansatz, der es ermöglicht, solche formalisierten Darstellungen zu verarbeiten. Das Verarbeiten formaler Darstellungen kann automatisiert werden, um Entscheidungen im Entwurfsprozess zu treffen, das Ziehen und Verarbeiten von logischen Schlussfolgerungen über das System zu unterstützen und Code-Generierung im Systementwurf zu ermöglichen. Der entwickelte Formalisierungsprozess kann in die folgenden Schritte aufgeteilt werden: Die ursprünglich in natürlicher Sprache dargestellten Anforderungen und Tests werden zunächst manuell (von einem Ingenieur) in logische Ausdrücke überführt. Hierzu werden bereits existierender 'Specification Patterns' genutzt. Logische Ausdrücke ermöglichen sequenzielle boolesche Umformungen, daher sind alle folgenden Umformungen mathematische beweisbar und somit korrekt. Die erste Darstellung in logischen Ausdrücken erfolgt in Form von Linear Temporal Logik. Diese Darstellungsform ist kompakt, aber nicht für jeden Zustand explizit, da zeitliche Abhängigkeiten in innerhalb eines Statements enthalten sind. Umformungen in Prädikatenlogik und Sortieren in Konjunktiver Normalform erzeugt eine zustandsweise explizite Normalform. Darstellung von Anforderungen und Tests in zustandsweiser Konjunktiver Normalform ermöglicht damit logische Schlussfolgerungen über Redundanz und Konsistenz eines Datensatzes von Anforderungen oder Tests. Die Darstellung von Anforderungen in einzelnen Zuständen kann in einen einzelnen Zustandsautomaten überführt werden. Dieser Zustandsautomat stellt damit das gesamte System dar und ist analytisch korrekt. Dies bildet die Grundlage für weitere Untersuchungen.

Die Konsequenz der Existenz eines solchen analytisch korrekten Transformationsprozesses ist, dass die Fragen nach Korrektheit, Konsistenz und Vollständigkeit eng mit einem Darstellungsproblem verknüpft ist. Die genannten Fragen können mit diesem Ansatz beantwortet werden, wenn eine initiale, teil-formale, schablonen-basierte Darstellung erzeugt werden kann. Die eingeführte Prozesskette ist mit einer schrittweisen Transformation anhand von Beispielen für Anforderungen und Tests dargestellt. Korrektheit, Konsistenz und Vollständigkeit von Anforderungen und Tests für ein System können mit einer Darstellung dieses System in Form eines Zustandsautomaten gezeigt werden. Diese Arbeit zeigt eine Lösung zur Transformation von logischen Ausdrücken in finiten Zustandsautomaten. Die Korrektheit und Anwendbarkeit dieses Ansatzes wird mit industriellen Fallstudien gezeigt. Die im Ansatz gezeigte formale Repräsentation ermöglicht es, die Fragen nach Korrektheit, Konsistenz und Umformung zwischen verschiedenen äquivalenten Repräsentationsformen zu adressieren. Zusätzlich wurden die durch Skalierung entstehenden Effekte in industriellen Anwendungen untersucht und diskutiert.

Zusammenfassend wird in dieser Arbeit ein Ansatz zur Formalisierung von Anforderungen und Tests eingeführt. Es werden Korrektheit, Anwendbarkeit und Skalierbarkeit des eingeführten Prozesses und der einzelnen Verarbeitungsschritte im industriellen Kontext gezeigt. Daher kann der gezeigte Ansatz als eine Lösung zur automatisierten Verarbeitung von Anforderungen betrachtet werden. Der Ansatz stellt eine Lösung dar, die Disziplin des ‘Requirements Engineerings’ in den Kontext des digitalen Produktentwurfs zu integrieren.

1. Introduction

> "Complexity increases when variety (distinction) and dependency (connection) of parts or aspects increase"
>
> Francis Heylighen [Hey99]

There is a trend in the automotive industry towards a rising number of components with a technological contribution to the car functionality. This is perceivable in technologies like autonomous driving, which is based on a strongly connected interplay between many hardware and software elements. Sensors such as lidar, radar and camera deliver input and feed a prediction algorithm based on a neuronal network. Neuronal networks are already in use for object detection in today's car while prediction algorithms are in development. Another example is the connected car in an *'internet of things'*-like surrounding with constant interaction between car, passengers, environment, other local cars and global traffic flows. Autonomous driving and connected cars are just two of many examples for this trend towards increasing technological complexity increase. In addition, customers expect to choose between many model variants and expect the car of choice to be customizable based on personal preferences. Heylighen [Hey99] defined complexity through a combination of two factors: Increase in variety and increase in dependency. Extension of model lines and the mentioned desire for customization affects the variety of designed and produced variants. This increase leads to bigger systems with increased dependencies. It is therefore fair to say that automotive development is experiencing a tremendous increase in complexity. In addition, pressure from other industries, e.g. mobile communications, requires faster technology introduction times. Shorter time-to-market causes development cycles to drop. This puts high demands on the automotive development processes and challenges existing development paradigms.

The state-of-the-art development process in the automotive industry is the V-model shown in Figure 1.1. It was adapted from the waterfall model, which can be seen as

the first formally described development process. It contains phases for specification, development and testing of a product. While the waterfall model creates a structure for all listed development steps, it lacks flexibility in case of changes and arising problems. For complex systems, the waterfall model is rather slow and static. It cannot adjust to today's challenges and required flexibility.

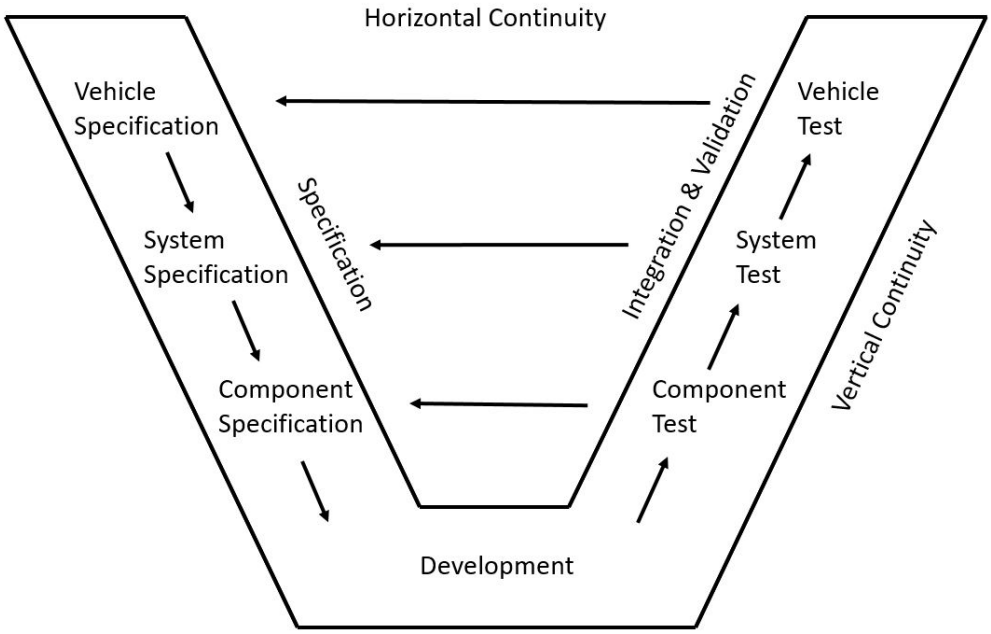


Figure 1.1.: Development Process: V-Model (MBC)

In contrast, the V-model (with the same lack of flexibility), while providing the same vertical continuity as the waterfall model, adds horizontal continuity. All levels (component, system and vehicle) create a link between tests and the corresponding requirements. While the V-model seems generally appropriate for the given challenges in automotive system development, the representation of its requirements and tests, as observed in current industrial projects, is clearly outdated. The standard as of today is the representation of requirements and tests in the form of textual (test) specification documents. This is not a particular characteristic. In industrial projects, the V-model and textual representation of requirements and tests are, in fact, bundled. Textual representations of requirements are easier to create and are needed in non-technical contexts (e.g. management, law, marketing), yet limited ambiguity and limited consistency of the representation structure prevents machine-based inference to a high degree. The increased complexity (more product variants and higher degree of inter-dependencies)

requires a more suitable representation form that can express the system complexity and allows computer-based assistance in the form of knowledge inference on the data set in use. Model-based representations fulfill the inference capability for requirements. While this resolves an existing problem, a new problem has to be addressed: the still needed textual representation and the model representation require consistency. The logical solution is a Text-to-Model (T2M) or Model-to-Text (M2T) transformation (see Section 3.2). Another challenge arises from the development model in common use of today for automotive systems. The state-of-the-art method in the automotive industry is that, after requirements are elicited and documented in a system specification, Original Equipment Manufacturers (OEMs) outsource part of their system development to suppliers and service providers. After the component development is finished by the supplier, the system must be integrated within the existing system. Simply said: OEMs specify systems but development is (in part) done outside the company and the OEMs only integrate and test the development afterwards. The number of participants, the challenges associated with correct interface definitions between systems, and the efforts to achieve consistency during system integration put high demands on (test) specification and system verification and validation. Therefore, the field of verification and validation is burdened by the same problems as the domain of requirements engineering.

This work is concerned with the challenge to provide a sufficient method to bridge the gap between textual and model-based specification. It assists automotive systems engineering in two ways: First, it supports development by providing a consistent conversion of textual requirements into executable system state machines. This supports model-based system development efforts. Second, it benefits the system integration process due to the formal representation of test specifications. The derived formalism allows detection of redundancy, inconsistency and unambiguity of test statements. Post-processing of detected redundancies significantly reduces test efforts by reducing overall test loads. Textual representation of requirements occurs mostly on the system and product level. Similarly, the integration problem of externally developed components and systems into a final product (system of systems) occurs on the system level. Therefore, the scope of this work is limited to the higher development levels, particularly the system level. The scope is further limited since model-based representations are suited for functional requirements but might lack the ability to represent Non-Functional Requirements (NFR). An average system specification at Mercedes-Benz Passenger Cars (MBC) contains about 2000 statements, of which about 1100 are functional requirements, 500 are NFR and 400 are head-

ings and information statements. Subsection 2.2.1 contains an in-depth discussion about requirements and the differentiation between functional and non-functional requirements. NFR and their tests commonly maintain their textual representation throughout the development process due to the non-existence of a more formal expression. Therefore, this work is limited to functional requirements and its tests. Functional requirements are validated through functional tests and integration tests. Empirically observed, an average system at MBC contains, as previously mentioned, about 1100 functional requirements where on average for one functional requirement about 1.7 test cases are designed [WHPR17]. The intention of this work is to improve such systems designs in regards to their functional requirements and test representations in order to assist specification and integration efforts.

1.1. Automotive Knowledge Representations

State-of-the-art in automotive system development at MBC is that the majority of functional requirements and tests on the system level are described in textual form. While different approaches exist to describe systems and their requirements via models (e.g.: state machines), as of today, the vast majority of system descriptions for requirements and tests at MBC exist in natural language form. Projects that use v-model-based approaches do not provide a consistent textual representation. There exists no widely used approach to maintain both forms (text and model) in a consistent coexistence without manual conversion and updating. At the same time, natural language expressions are prone to contain subjective meaning. Natural language is ambiguous. For knowledge representations, specifically in the domain of engineering design, this turns out to be problematic. One approach to reduce this problem is grammar limitation (reduced syntax) in combination with a reduced description vocabulary. Knowledge represented in natural language is transformed through surjective mapping into limited grammar representation in the *limited grammar and syntax* space as shown in Figure 1.2. Because the mapping occurs in a non-mathematical domain (language space), rules and operations that apply in mathematical domains cannot necessarily be used to perform or validate the mapping. This is a language-related problem which is beyond the scope of this work and therefore will not be discussed further. Such a representation will from now on be called *structured language*.

From structured language representation and its language space, expressions can be transferred to the mathematical space, specifically logic space (see Figure 1.2). In that space, objective rules apply and mathematical operations can be performed. System descriptions may contain time-dependent elements represented in temporal logic. By applying time-related boundary conditions, these elements can be represented in a time-discrete form. For such a representation, the number of operators can be reduced. All time-dependent operators are obsolete and only time-independent operators remain in the local representation. This reduction is a surjective mapping from first order and temporal logic space to First Order Logic (FOL) space. For an unambiguous representation form, the knowledge can be represented in a standardized form. This standard form opens up the possibility for a formal system representation that allows knowledge inference and general machine-based data processing. The described transformation from natural language to restricted grammar, mapping to logic space and reduction of temporal elements towards a time-independent local representation is the core principle derived and discussed in this work.

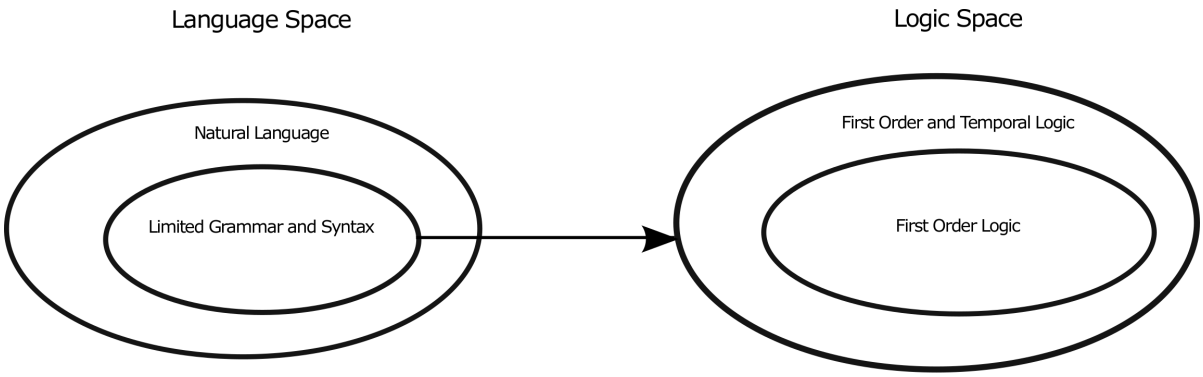


Figure 1.2.: Knowledge Representation: Language and Logic Space

The practical implementation of such a formalization model can be realized through a central graph-based data model. To derive such a model, existing data has to be preprocessed. Two forms of preprocessing shall be distinguished in this work: Preprocessing of the modeling structure and content processing via description formalization. Subsection 2.1 discusses modeling structure while Section 2.3 explains the principles of data formalization in detail. The raw data for general automotive system descriptions consists of a decentralized collection of documents. These documents contain system-related data sets including requirement and test descriptions. All data is stored in a requirement management tool in a tool-specific data format. System modeling must ful-

fill two needs. First, data has to be transformed into a tool-independent data format. This allows use of a variety of tools and secures data accessibility for a long time period. (This es becoming increasingly relevant, for example in context of the functional safety, where car manufacturers are required to provide long-term proof for certain system development and testing procedures to ensure quality standards). In addition, an integral, central data model provides further advantages in terms of data optimization. This generates visibility for data dependencies, increases traceability and simplifies data re-usability (manipulation, deriving domain-specific and tool-specific models). Graph-based design languages are used for data modeling. The building blocks of graph-based design languages are defined axioms (vocabulary) and execution rules (data manipulation). Starting with one axiom, rules are applied in sequential order to manipulate the axiom and its mutation. The result can be represented in a design graph. Such a design graph therefore contains all design decisions in an abstract form. The abstract representation can generate domain-specific models for domain-specific analysis.

1.2. Objective and Vision

This work intends to provide a formalization method for requirements and test representations to improve requirements management methods in terms of requirements engineering. The approach intends to solve two problems: It bridges the gap between textual and model-based requirement expressions during specification and downstream development steps. In addition, formalization for tests improves system integration efforts by reducing redundancies, inconsistencies and ambiguities in test expressions. This significantly reduces test loads and supports the system integration process.

Any form of data formalization requires an underlying ontology for data representation. The ontology used in this work is a so-called graph-based design language. Graph-based design languages in Unified Modeling Language (UML) can be used to generate fully (or partially) automated product designs. Rudolph et al. propose one particular approach [AR03, AR04] which includes an underlying graph-based design methodology and a software called Design Cockpit 43 (DC43) [IIL17a, IIL17b]. This work intends to show how formalization of system descriptions, particular requirements and test data can support industrial system design. To achieve this, the goal of this work is to derive a requirements and test formalization methodology consistent with the given graph-based design approach by Rudolph et al. [AR03, AR04]. The developed methodology in this work

allows system specification engineers to specify requirements in structured textual form and derive consistent model representations. Adjustments to the requirements lead to changes in the model which can be observed in real-time during the specification phase by executing the derived model. In order to prove the correctness and usefulness of the derived models, case studies are performed and evaluated at MBC development. Similar to the improvements for requirements engineering, the proposed approach can assist test efforts during system integration. The formal representation of test cases and its test steps enables machine-based analysis of redundancies, inconsistencies and ambiguities for test data. Particular the occurrence of redundant test steps is investigated and evaluated in industrial case studies. This includes a heuristic post processing to remove redundant test steps inside a forward directed chain of test steps. A case study evaluates the heuristic approach and shows potential a test load reduction. Overall, requirements engineering and system integration efforts will be assisted by a formalization method.

The vision for this work is that the derived formalization approach bridges the gap between (informal) requirements engineering and digital product development via a manual transformation step. The conversion T2M shown in this work provides a basis for model-based systems engineering and model-based product design. The Model-to-Model (M2M) transformations make it possible to derive a model of the digital product based on its (formalized) requirements. Therefore, the approach described in this work, leads the way to a formal and closed process chain for digital product development along the V-model. Consequently this leads to the vision of a 'machine-executable V-model'.

1.3. Contributions

The main contribution of this work is the creation of a formalization process chain for (automotive) requirements and tests. The creation of the formalization process chain is achieved by combining three separately preexisting approaches into one:

- Specification patterns [DAC98] (conversion of language patterns to temporal logic)
- State-wise representation of linear temporal logic into first order logic [AF00]
- Transformation of logic expressions to state machines [LL12],[KVBSV12]

Each approach existed in isolation for a subset of the overall problem. The combination of these approaches is performed in this work and leads to a single overall approach.

Further, this approach is validated against industrial data from MBC. It shows the reduction in time effort and effort needed for specific requirements engineering tasks. In addition, it addresses the possibility of automated data analysis.

1.4. Structure

This work consists of three core chapters and two framing chapters. The three core chapters are a chapter about the theoretical background, a chapter on the new modeling approach developed throughout this work and a chapter on industrial applications to validate the model. These three chapters represent the center piece of this work while the introduction and summary tie the overall idea and approach together.

Chapter 2 discusses the theoretical background and state of the art of the field. It is separated into three blocks: Section 2.1 discusses the methodology and general idea of graph-based design. This is used as the underlying modeling structure of the later derived model. In Section 2.2, the state of the art for requirements engineering and testing is shown. This is the particular industrial field in which this work takes place. The third block consists of different forms of knowledge representation. Section 2.3 lays out all knowledge representation forms relevant during the data processing as well as two brief discussions about determinism in systems and modeling structures.

Chapter 3 derives a new approach to formalize requirements and test data. This chapter is the center piece of the work. Section 3.1 provides the big picture with an overview of the model. The following sections show and discuss the particular conversions between representation forms. Section 3.2 addresses the mapping between natural language or textual representations and structured textual representations in the form of language patterns. In Section 3.3 mapping from language to logic space, particularly to temporal logic, is shown. The next two sections discuss mapping from temporal logic to simple and more complex structures. In Section 3.4, mapping for forward-directed chains is shown. This is required to model and formalize test data. Section 3.5 is concerned with the generalized case, e.g. mapping from temporal logic onto cyclic directed graphs. Such structures mimic state machines. This allows requirements to be modeled and formalized. In the last section, the transformation of local FOL representations into a normal form is presented. This is shown in Section 3.6.

Chapter 4 provides four case studies to validate the derived model. Section 4.1 validates the model for test cases which represent the most simple structural form, a forward connected chain. To apply the model towards test cases in industrial contexts, a post-processing is necessary. This is covered in Section 4.2. These two sections show the potential improvements that can be achieved by the derived model in the field of system testing. In Section 4.3, the approach is validated for more complex structures, in this case, cyclic directed graphs. Requirements represented as finite state machines can be represented in such structures. This application is the core application of this work. The initially static finite state machine can be extended towards an executable state machine. Section 4.4 extends the derived approach and makes it possible to dynamically control the derived state machine.

Chapter 5 summarizes this work and closes with a conclusion. Relevant results are discussed and put in context. The derived formalization method and its industrial applications are evaluated in Section 5.1. Limitations of the approach are addressed in Section 5.2. Section 5.3 provides a brief outlook on related fields, further potential research topics and remaining open questions.

2. Background and State of the Art

> *"The answer to the ultimate question of life, the universe and everything is forty-two."*
>
> Douglas Adams,
> The Hitchhiker's Guide to the Galaxy

The goal of this work is *"To derive a formalization process for requirements and test data and to apply it to electric/electronic Systems (E/E Systems) at Mercedes-Benz passsenger car development"*. This shall be done to reduce the required time effort for requirements engineering and to allow automated analysis of the data. In order to achieve such a formalization process, existing principles and state-of-the-art work shall be considered. There are three primary topics that build the foundation of this work: Graph-based design as the ontology, automotive systems engineering in regards to existing requirements engineering and testing, and (selected) knowledge representation forms for the field of system engineering.

Graph-based design in Section 2.1 introduces the underlying principles of the data model. It includes a general discussion about the principles of design languages and outlines in detail the approach used in this work to create, update and optimize the data model. The work takes place in the context of requirements engineering and testing. Section 2.2 discusses the state of the art for systems engineering and, in particular, requirements engineering and testing (verification and validation). This includes specific references to the development model used, the definition and differentiation between forms of requirements and the definition of test-related terms. The premise of this work is data optimization through inference from formalized data. Data formalization for inference is rooted in (formal) data representation. Section 2.3 includes an in-depth discussion of the four representation forms used in this work. It is evaluated why natural language is used as the state-of-the-art representation form and what problems arise from that.

Different forms of specification patterns are introduced. It is shown how systems can be represented in the form of state machines. In addition, two particular problems (deterministic versus non-deterministic) and modeling structures (chains versus divergent forms) are discussed.

2.1. Graph-Based Design Assumptions

Graph-based design as a discipline is not naturally connected to requirements engineering. When requirements are represented in a formal representation form, an ontology is needed. Therefore, in this work, graph-based design is used as the ontology for the representation of the formalized data. Designing and building products is a difficult task. It is obvious that in order to build the best or at least the most suitable product, eventually other design solutions have to be abandoned. The question is now rather: "How do we assure that the most suitable solution becomes visible to us?" Design starts with the whole solution space where in principle all designs are valid solutions. Every design decision limits the solution space and with that the number of potentially valid solutions is reduced. This continues, eventually to the point where *"the most suitable"* solution lies outside the solution space. This happens not because of the wrong assumption that the best solution, is actually not the best solution but because the solution space is constrained in an incorrect order (or with an unwise prioritization). To avoid such a scenario, it is essential to select the order of design decisions carefully. It must be analyzed which design decisions are necessary or inevitable (e.g. physical laws) and which decisions are secondary because they are only based on personal taste or subjective metrics. A solution to the prioritization of design choices can be derived from the concept of abstraction.

"It is a fundamental idea in Piaget's work on the development of logical and mathematical thinking that ideal objects such as abstract mathematical ideas have their origin in the closure of such systems. Once a system is closed, the properties of the constituted ideal object are no longer dependent on empirical experience. Rather, they are determined by the structure as a whole and can be elaborated by the operations of logical and mathematical thinking", as quoted by Damerow [Dam96] based on Piaget [Pia52]. The term closure in this context represents an assumed and established solution space. Piaget points out that mathematical abstraction removes real world object influences, perceived through empirical experience from the system described or envisioned. Removing certain aspects increases the focus on the remaining aspects. Abstraction directs

attention to underlying inherent structure, dependencies and limitations put on a design by physical laws and mathematical constraints. It becomes obvious, that in order to prioritize design choices, an abstract representation can lead to the *true* solution space. True solution space here is defined in the sense of a formally complete design space. Such space remains once the initial solution space is reduced and limited by considering underlying, non-negotiable constraints. It reveals the true solution space in which design decisions, based on a personal chosen optimization metric (or simply personal taste) can be made. To derive this true solution space, a design approach with a degree of abstraction seems suitable. One such approach are string-based, shape-based and graph-based design languages. Design and creation take place in an abstract form. Interpretation of symbolic representations performed by a compiler turns abstract symbols into concrete design objects. While the abstract design approach includes underlying constraints and, with that, limits the solution space, it reveals the left-over solution space (here called *true solution space*). This true solution space is consistent with all non-negotiable constraints. An example could be the use of a certain part which is made of a specific material. Any material has a temperature range in which it can function properly. Therefore, the solution space is limited in the temperature dimension. All designs inside this space are valid and can be chosen from. In Subsection 2.1.1, the general principles of design languages are investigated, while Subsection 2.1.2 shows the particular approach that is used in this work. Subsection 2.1.3 takes a first step in assessing how graph-based design can support the task of improving requirement and test data.

2.1.1. Graph-Based Design Languages

It was discussed that abstraction includes the idea of revealing the underlying structure or functionality of a design. Another aspect of abstraction is therefore analogy. Two initially seemingly different objects share commonalities that are revealed when analyzing the abstract form of its inherent structure and functionality. One potential engineering approach is to achieve design solutions by reasoning from analogies between the design envisioned and an existing solution observed in another, already existing domain (or product). The solution from the implementation layer in one domain can be abstracted. In the abstraction layers, both domains share the abstracted problem and its abstract solution. This analogy can be used to transfer the existing solution from the initial domain to the new domain, where it is transferred to the implementation layer.

One prominent occurrence of this principle is the field of biomimetics, where solutions observed in nature and biology are abstracted from and transferred to engineering. Often times biology provides solutions that seem unintuitive to engineering design. Problem solution in engineering is most often approached by decomposition from high level to more fine grained structure and functionality. Such a top-down approach is in contrast to the incremental bottom-up approach in nature and biology. Combinations of basic building blocks create more complex entities with higher functionality. Simple mutations create design variants and natural selection (Darwin [Dar68]) evaluates the variants. Underlying the principle of bottom-up approaches is the fact that each variant at every moment has to be in a stable state. This means that e.g. mass, energy and static equilibrium must exist at any moment. A top-down approach can (potentially) be in an unstable state at any moment until finished. Difficulties for top-down approaches lay in the fulfillment of requirements. It is necessary to refine requirements into more low level requirements and divide a product into meaningful parts so that every part can fulfill a subtask that contributes to the overall task. Therefore, *complexity lies in the decomposition.*

For bottom-up approaches, the opposite is the case. The challenging task here is in the fact that the composition of parts has to lead to a meaningful, functional product. Therefore, for bottom-up approaches, *complexity lies in composition.* Classic engineering prefers a top-down approach to fulfill (high level) requirements. As shown, complexity here lies in decomposition of overall functionality into controllable fractional parts of functionality. The same level of complexity occurs again, once the envisioned parts are integrated. Because a moderate increase in size for each part leads to an exponential increase in interfaces, this process experiences a tremendous (exponential) rise in overall complexity. Complexity increases to a level where it is questionable whether such an approach is feasible in the future without adjusting affected manual process steps and automating these to handle the increase in work load. Another potential path out of this problem is bottom-up approaches where complexity roots in a totally different area. The challenge is to assure that composition actually solves the given problem. Therefore, the process of composition has to be steered and guided in the right direction. This problem is discussed in Subsection 2.1.1 and Subsection 2.1.3.

One form of a bottom-up approach in engineering are formal language grammars. The combination of a number of primitives (building blocks) and a set of rules creates a so-called production system. This allows for the generation of syntactically correct sen-

tences belonging to this grammar [AR03]. Formal language grammars occur in the form of natural language, programming languages and other forms (e.g. L-System grammars by Lindenmayer and Prusinkiewicz to describe plants [PL12, PH13]).

The construction of sentences occurs by using and combining vocabulary (building blocks) in ways defined in execution rules. This creates syntactically correct sentences. To create a semantic meaning, an interpretation for each symbol is performed, where each symbol is assigned a particular meaning. The idea of a formal grammar with its underlying production system can be abstracted. Higher forms are shape grammars where building blocks are lines and shapes instead of words [Sti80, GS80]. Another form of design grammars are graph-based design languages [SR16, AR03, AR04]. The abstraction level is increased further in such a way that the building blocks are generic nodes. Rules connect these nodes, generating a so-called *design graph*. The design graph is abstract and its symbolic interpretation becomes a key process step. The same graph, based on its interpretation, can represent different entities in different domains. The generic gestalt of a graph in graph-based design, opens a variety of application fields (different domains, multi-domains and increased complexity). Alber et al. [AR03] describe the representation of information for each entity and the ordering structure as the two requirements for a design language. A graph can hold structure, ordering and hierarchical information while symbolic interpretation converts information stored as an attribute of a node into meaningful forms. These characteristics make graph-based design a suitable approach for the given problem set.

2.1.2. Graph-Based Design Language Applications

In Subsection 2.1.1, the concept of design languages and grammars was explained. Graph-based design languages are the most abstract form of design grammars. Their purpose is the creation of complex systems in a reproducible, traceable way. At first a production system has to be defined. A production system contains primitives (building blocks) and a rule set. Building blocks are instantiated, sequentially combined and evolved from primitives into increasingly complex entities through a given set of execution rules. Each execution rule represents a design decision and is applied based on its call in the sequence diagram. This is explained more in detail later. Figure 2.1 shows a rule set with generic rules. An example for a specific rule would be the design decision to always place a damper between a wheel and the steering axis. In rule 3, the white

node represents the abstract wheel, the shaded node represents the abstract damper and the filled node stands for the abstract axis. The left side shows the 'before' (without damper), while the right side shows the 'after' (including damper).

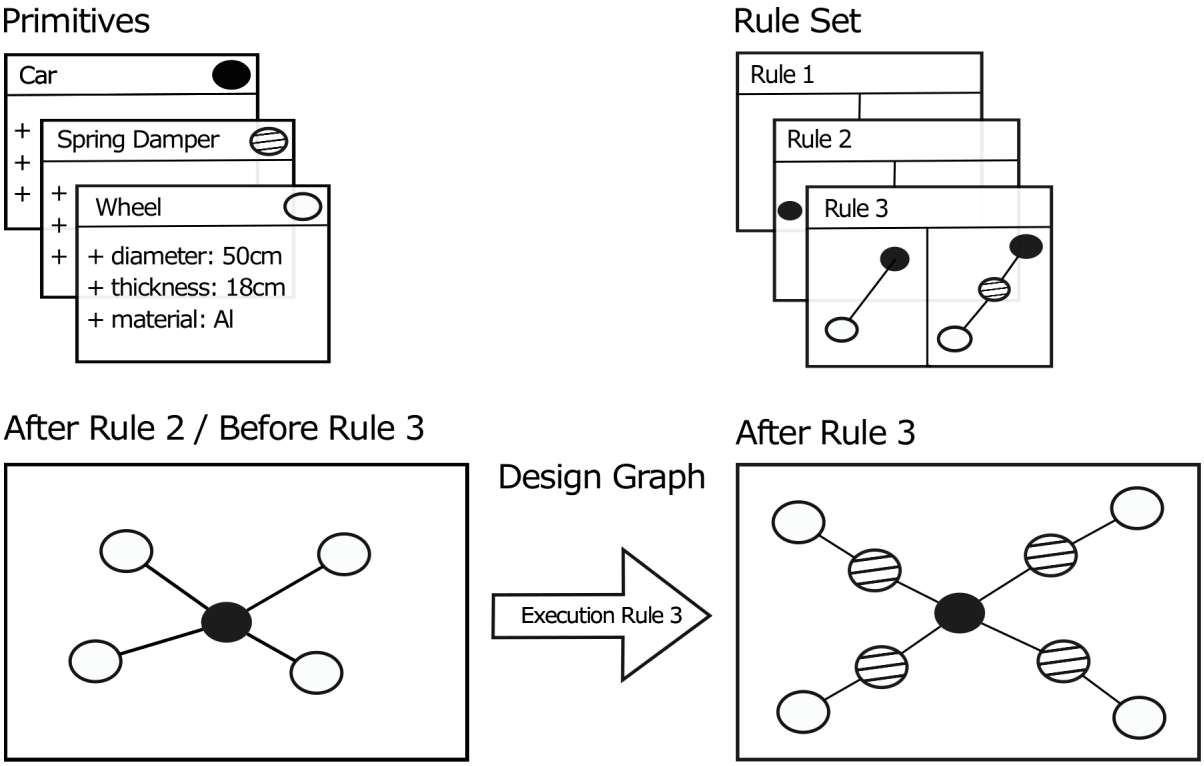


Figure 2.1.: Graph-Based Design: Primitives, Rules and Design Graph Adaptation

Instantiated elements and their progressions are represented in the design graph. Each element is represented by a node and its dependencies by links. These principles are illustrated in Figure 2.1. Each primitive element and each generated complex element hereby can serve one of two functions at a time. First it serves as a new primitive where it can be potentially evolved further through additional execution rules. Rule execution is followed by compilation. Generally, each entity serves in its second function as a syntactical placeholder for an element with a corresponding meaning. Mapping from element as placeholder to element representing a syntactical meaning is called *interpretation* and is achieved during compilation. This principle allows grammars and its corresponding graphs to represent entities in all kinds of (engineering) domains. Graph-based design languages require the creation of a domain-specific definition of primitives and rules. These can be stored in a domain library. The creation of more and more libraries makes

graph-based design a widely applicable system design approach and opens up a variety of opportunities for the design of complex systems.

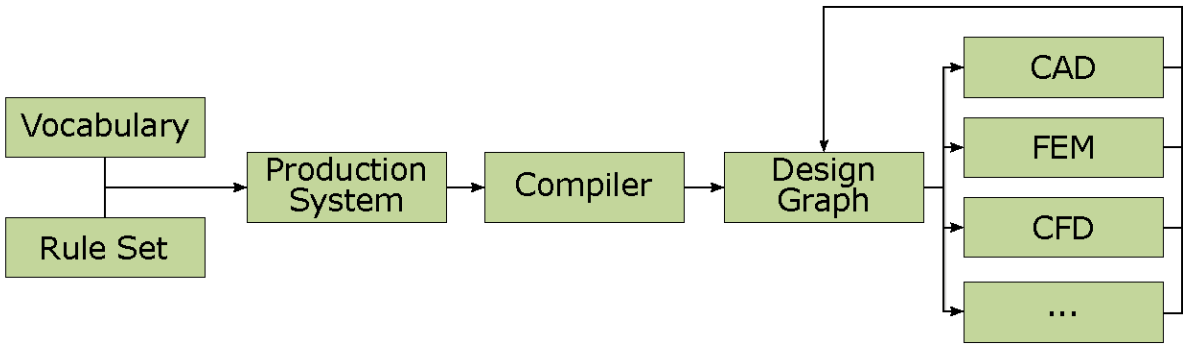


Figure 2.2.: Graph-Based Design: Process Overview, Rudolph [Rud06]

An existing problem in design is the natural opposition between domain-specific expressiveness and generality. Domain-specific design can describe one particular domain well. It uses domain-specific vocabulary (building blocks) to optimize a design variant in the particular domain and the specific design space. The problem here is, that overall consistency for all design domains is required. Given the fact that each domain can have a unique design space, it is intrinsically not guaranteed that there exists a design variant with overall consistency for all domains. The obvious solution to achieve overall consistency therefore would be a shared design space with a general model. Yet, the generality which allows for consistency, is too generic for domain-specific problems and thus limits the general model in terms of its domain-specific expressiveness. Solution to this dilemma is a combination of a shared general model with derivatives to all particular domains. Given that principle, it can be said there are three overall constraints that a design language has to satisfy in order to be capable of representing a sufficient solution for designing complex systems:

- A given need for a *'step-wise rule-based inference.'* [AR03]
- *"Second, it is necessary to represent the information which is needed to directly define a design object in a way which makes afterwards the necessary mapping from the symbolic representation to a semantic processing of the description as easy and straightforward as possible."* [AR03]
- The requirement that a general model achieves a consistent generic design with the capability to derive domain-specific detailed models with increased expressiveness.

One such approach is the graph-based design methodology by Rudolph et al. [AR03, AR04, SR05, SR16]. The core principles are illustrated in Figure 2.2. Step-wise rule-based inference is achieved through the production system, consisting of vocabulary (primitive building blocks) and rule set (execution rules). The compiler provides the capability for mapping from symbolic representation to semantic form. Its output is a design graph which represents the design and all design information in a generic form. Domain-specific models can be derived. Adjustments in the domain-specific graphs are fed back to the generic design graph through *'round trip engineering'*.

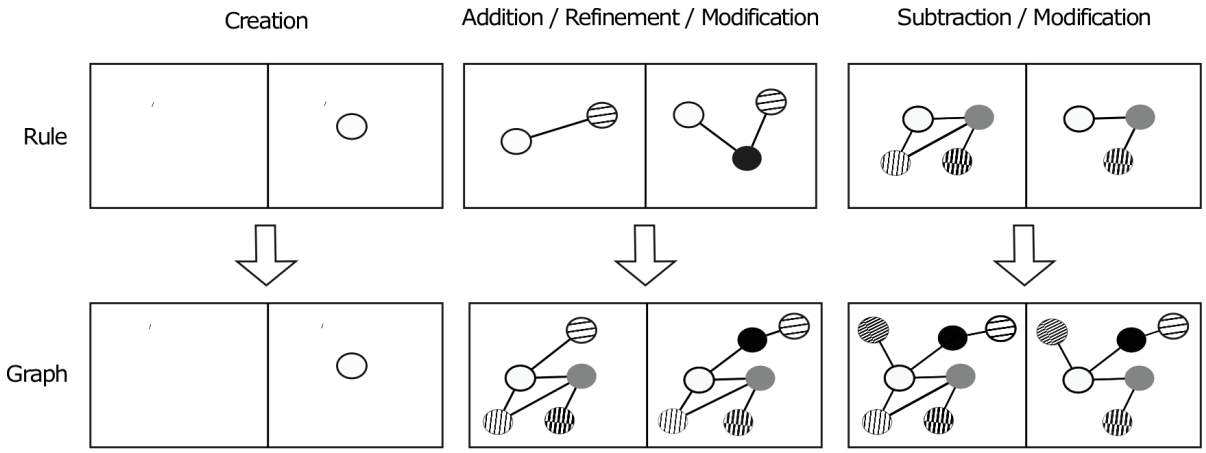


Figure 2.3.: Graph-Based Design: Design Graph Adaptations

One software application for the shown methodology is DC43 [IIL17a]. DC43 represents the abstract form of all primitives in a class diagram. This includes class attributes, linking and inheritance of relationships to other classes. A sequence diagram contains the rule set which instantiates objects from the given classes provided in the class diagram to the design graph. These objects serve as the primitives and are manipulated through the rule set. Each rule either consists of code (java rules) or is graph-based. Instantiated objects and relations form a graph (design graph). Graph-based rules work by the principle: Left-Hand Side (LHS) plus Right-Hand Side (RHS) (see: Figure 2.1 - Rule 3). Alber [AR03] lists three forms of graph-based modification rules (Addition, Decomposition and Modification). In this work, two additional forms are added to the listed to show the complete set of possible rules (Creation and Subtraction). Figure 2.3 illustrates the five rules while the following lists explains the setup and cause for each:

Creation

Setup: Empty LHS; entities at the RHS.

Cause: Entities occurring on the RHS are created. This can be seen as a special form of Addition.

Addition/Embedding

Setup: Entities (and potentially links) on the left-hand side; the same entities and additional entities and/or links at the RHS.

Cause: For every occurrence of the LHS pattern, the addition on the RHS is added to the relevant part of the design graph.

Subtraction

Setup: Entities (and potentially links) on the left-hand side; an identical set of entities and links, reduced by one (or more) links or entities on the RHS.

Cause: For every occurrence of the LHS pattern, the reduction between LHS and RHS is performed on the relevant part of the design graph.

Decomposition/Refinement

Setup: One entity on LHS; identical entity and additional entities with links to the existing entity on the RHS.

Cause: An entity is refined by additional entities. Special case of addition.

Modification/Adaptation

A combination of addition and subtraction rules to modify the design graph.

A rule is applied in the way that a certain configuration of instantiated objects and links is searched for as the LHS. Once found in the overall design graph, it is manipulated to what is shown at the RHS. The sequence in which objects are affected by the rule might be relevant; if so, this must be encoded in the rule as well. Each rule is applied once but can be applied in repetition when put in a loop with a termination condition in the sequence diagram. Executing and applying rules is an iterative process which eventually terminates, once all rules in the sequence diagram have been processed. If in certain cases termination does not happen, the rules or the sequence diagram have to be adjusted until termination is given. This is comparable with an infinite loop in coding. Aside from the rule set, further information like physical laws, mathematical constraints and boundary conditions are processed. This information is either stored in libraries (e.g.

physical laws and boundary conditions) or in the class diagram (e.g. problem-specific constraints and design-related equations). All these build a set of linear equations which can be solved by the solution path generator. This can lead to no, one or many solutions depending on the type of linear equation system (see Figure 2.4 bottom left box).

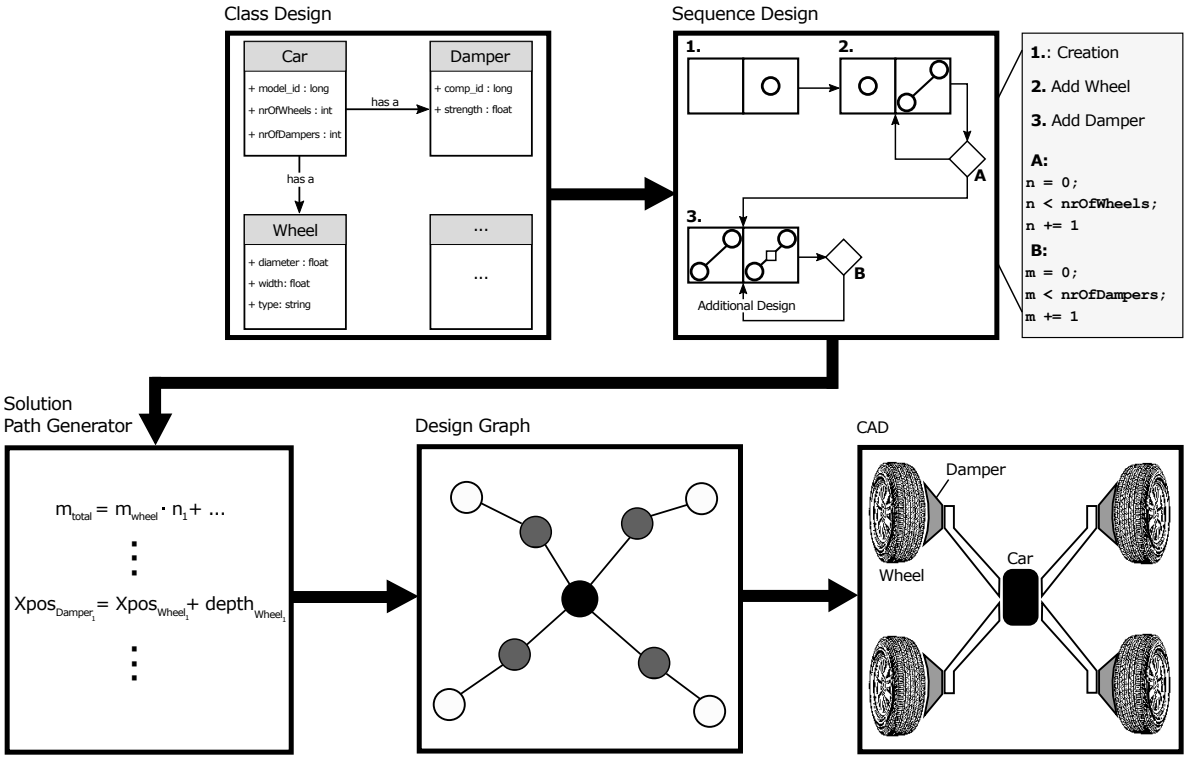


Figure 2.4.: Graph-Based Design: System Design (Example)

After solving the set of linear equations, the compiler creates the generic design model in form of a design graph. Interfaces allow the generic model to be derived into the specific domain models (Computer-Aided Design (CAD), Fintite Element Analysis (FEM), Computational Fluid Dynamics (CFD),...). In each specific domain, the abstract form of the design graph is mapped to its corresponding syntactical meaning. (e.g. in the form of geometry, stress and load forces or aerodynamic characteristics.) The domain model can be optimized further and information and adaptations are fed back into the generic model. This process was mentioned earlier as 'round trip engineering'. Such optimization is an iterative process between optimization of domain model and graph adaptation. Figure 2.4 represents the five core elements of graph-based design as approached in DC43.

The class diagram contains all classes with attributes and relations. The sequence diagram contains the rule set which is sequentially executed. Class diagram and sequence diagram are both initially defined in UML and adjusted to the graph-based design approach. The solution path generator processes all underlying constraints and equations. The generic domain-independent model in the form of the design graph is created and can be derived into domain-specific models. The general principle of modeling data in such a graph-based process with the support of DC43 is used in this work. Subsection 2.1.3 addresses how this can be adjusted to the field of requirement and test data.

2.1.3. Requirements and Tests in Graph-Based Design

In the discussion about the top-down versus bottom-up approach it, was shown that a downside of bottom-up approaches is the general problem of steering design towards the required or envisioned product functionality. The empirically observed situation for requirements elicitation and refinement for graph-based design, in particular DC43, shall be discussed here. This might not be representative for all graph-based design approaches but since DC43 is used in this work, this particular situation shall be laid out.

Graph-based design in this context serves as the underlying ontology to represent and process the requirements and tests throughout the formalization process. The problem to be solved here is that graph-based design usually uses a bottom-up approach while requirements engineering in a V-model refines requirements top-down. Requirements are initially defined on a high level (product or full system requirements, e.g. *"The car must run at all commonly accepted speeds."*) and then defined with more detail later on (e.g. *"The maximum temperature of the engine shall not exceed the maximum working temperature of the material used for building it."*.) Thus, requirement refinement is not naturally aligned with the bottom-up approach of graph-based design. Subsection 2.2.1 divides requirements into functional, performance, quality and constraints. It is observed that refinement for constraints is intuitive when designing with DC43. Classes are constrained through boundary conditions of attributes in exactly that way (e.g. $maximum\ car\ mass = 2300\ kg$). Therefore, a need for mathematical equations and constraints exists naturally. Problems arise with the other three forms of requirements (functional, performance and quality). These requirements are usually complex to refine. It is perceived that DC43 designs usually work with high-level requirements while refinement for low-level requirements takes place informally through design decisions. This creates

implicit assumptions and knowledge about the design and the product, without any formal documentation or process. Such an approach is problematic for a variety of reasons (requirements validation, design adjustments, hand-overs, ...).

Chapter 3 shows a path that solves this problem by connecting requirements elicitation and refinement with the DC43 design process naturally. Background and relevant definitions in the field requirements and tests are given in Section 2.3.

2.2. Automotive Systems Engineering Methodology

The Mercedes-Benz E-Class released to market in 2016 contains about 200 systems. For theses systems, complexity varies tremendously. Personal observation estimates the average system size at about 1500 requirements and 2500 test cases (system level). Product complexity requires a systematic development approach. This section introduces Product Development Process (PDP) as described in Pahl & Beitz [PBSJ13, PBFG14]. It is aligned with the V-model, state of the art model in the automotive industry today. Special focus is on requirements engineering (Subsection 2.2.1) and verification & validation (Subsection 2.2.2). To define the scope of this discussion about system engineering, two questions should be answered upfront:

What is a system (and a system of systems)?

What are the disciplines and tasks required to develop a system?

One answer to the above questions is within NASA's definition for system engineering:

Definition 2.1. *"**Systems engineering** is a methodical, disciplined approach for the design, realization, technical management, operations, and retirement of a system. A "system" is a construct or collection of different elements that together produce results not obtainable by the elements alone." [SA95]*

The term 'results' NASA uses in its system definition is equivalent with the expression 'functions' used by Pahl & Beitz [PBSJ13] in its definition of product:

Definition 2.2. *"Every **product** serves the purpose of fulfilling a function" [PBSJ13, pp. 242, translated].*

The definition of *system* given by NASA shall be extended in the sense that big systems (e.g. a car) are often separated into sub-systems, thus a car is a system of (sub-)systems.

All further discussions apply to systems and to systems of systems in the same way, thus they are not differentiated in this work. Yet, all non-technical domains like market analysis, marketing & sales strategy and customer satisfaction are beyond the scope of this work and and will not be considered further. [PBSJ13] further states: "A solution for a technical task including energy, material and signal transfer requires the existence of an unambiguous, reproducible relation between input and output of a system." [PBSJ13, pp. 242, translated]. This leads to the conclusion that: "A function is the common and sought relation between input and output of a system with the goal to fulfill its purpose" [PBSJ13, pp. 242, translated] (see Figure 2.5).

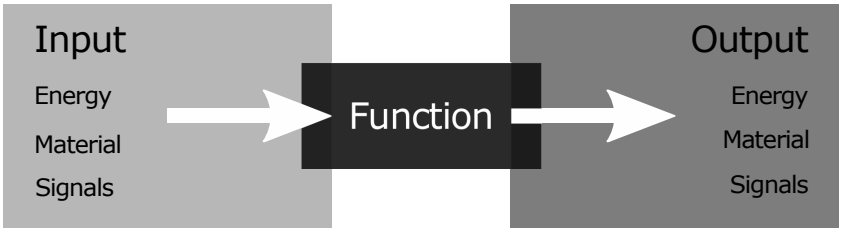


Figure 2.5.:
Product Development Process: Function, Pahl and Beitz [PBSJ13, pp. 240, trans.]

Functions can be further divided into sub-functions to reduce complexity. (Sub-)functions can be realized through physical effects or the combination of physical effects (work dependencies). "To achieve the overall functionality, work dependencies of all sub-functions are combined. The combination of multiple work dependencies leads to the work structure of a solution. Through a work structure, the interactions between work dependencies are identified. It shows the solution principles that realize the overall functionality." [PBSJ13, pp. 249, translated] Functions require physical implementations. Components serve as the physical structure for functional implementation. Usually functionality is not implemented on one isolated component. Components in general consists of a control unit with a CPU accommodated by an input layer (sensor or signal input) and an output layer (actor or signal output). Function realization often requires contributions from multiple components. This causes complex dependencies between functions and components.

The state of the art in the automotive industry for system development is the V-model (see Figure 1.1). It separates the development process in three major phases: Requirements specification with refinement from system to low-level (including architectural design), development and verification & validation including the system integration from low-level to high-level.

V-model and Pahl & Beitz's PDP can be mapped as shown in Figure 2.6. PDP and V-model define the system at first. Requirements (≈ functions) are specified in detail based on demanded system purpose. While there exists no formal solution generation in the V-model development phase, the PDP provides a formal sequence of steps to reach the demanded functionality. For all specified functionalities, solution principles are selected. Solution principles are capable of physically transforming the system and are implemented onto components. Functionality requires the interaction of components. Integration including validation and verification of correct behavior against specification, is the last phase of the product development. Components are aggregated to modules and modules are aggregated to a system. It should be noted that since the Pahl and Beitz approach in Figure 2.6 only contains one requirements specification phase, namely an exact mapping to the V-model, this specification phase must contain the high-level and low-level phases of requirements engineering.

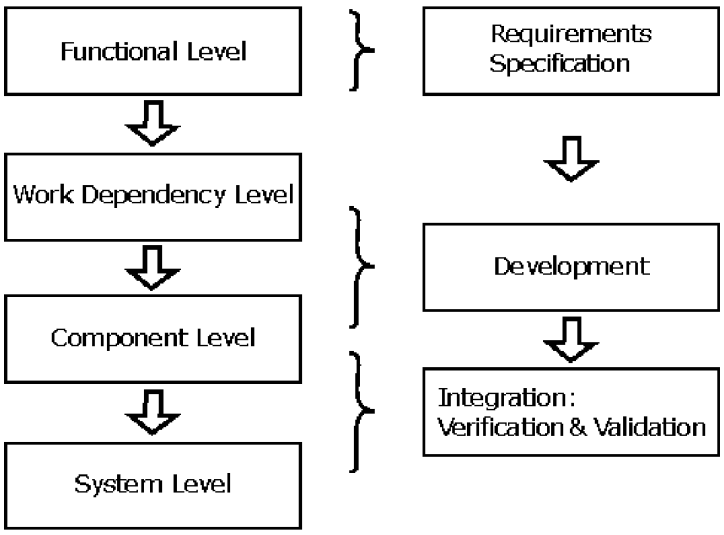


Figure 2.6.:
Product Development Process (PDP), Pahl and Beitz [PBSJ13, pp. 240, translated]

The scope of this work for requirements specification as well as integration including verification and validation is on the system level. System development is not discussed in more detail. Pahl and Beitz was introduced to provide a connection between requirements engineering and testing to the overall product development. There exist formal approaches (like the one shown) that allow requirements engineering and testing to be embedded into overall system development. Subsection 2.2.1 contains a detailed introduction to the state of the art for requirements engineering while Subsection 2.2.2 covers integration and related testing.

2.2.1. Requirements Engineering

In systems development, the first technical phase is concerned with the expression of all required functionalities of the product envisioned to be built. This and additional tasks take place in the domain of requirements engineering. The definition of requirements engineering used in this work is provided by Rupp and Pohl:

Definition 2.3. *"**Requirements engineering** is a systematic and disciplined approach to specify and manage requirements [...]" [RP15].*

The process of requirements engineering can be divided into four main phases: Elicitation & analysis, documentation, verification & validation and management of requirements. The future discussion mostly addresses the documentation process. For this purpose, it is necessary to define the term requirement and its differentiation into functional and non-functional. This provides the basis for a detailed discussion of requirements documentation and representation forms. Zave and Jackson [ZJ97] point out that "[...]it is not necessary or desirable to describe (however abstractly) the machine built". Jackson [Jac17] further concludes "Different possible behaviors can satisfy the requirements just as more than one possible program can compute a desired result. So the requirement [...] should not specify the behavior [...] directly, but only its desired properties, effects and consequences." Requirements therefore provide a rather abstract descriptions of the behavior and system capabilities instead of an exact system design. The term requirement from now on will be defined and used as:

Definition 2.4. ***Requirement**: "A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents." [IEE90]*

Requirements can be separated into Functional Requirements and Non-Functional Requirements (NFR). While there exists a wide agreement on the definition of functional requirements, "[...] there is still no consensus on the nature of non-functional requirements and how to document them in requirements specifications" [Gli07].

Glinz removes the term NFR in his taxonomy of requirements. He considers performance ("restrictions about timing, processing or reaction speed, data volume, or throughput") and quality requirements ("a specific quality that the system or a component shall have") as attributes of the described system,"... while any other restriction about what the system shall do, how it shall do it, or any prescribed solution or solution element" is considered a constraint. He remarks that it is possible to group performance requirements,

quality requirements and constraints together as NFR. The definition proposed by Glinz is the one used in this context while, as proposed, grouping all requirements other than functional requirements as NFR.

Definition 2.5. ***Functional requirement****: "A requirement that specifies a function that a system or system component must be able to perform." [IEE90]*

Definition 2.6. ***Non-functional requirement****: Non-functional requirements summarize performance requirements, quality requirements and constraints. "A performance requirement is a requirement that pertains to a performance concern. A specific quality requirement is a requirement that pertains to a quality concern other than quality of meeting the functional requirements. A constraint is a requirement that constrains the solution space beyond what is necessary for meeting the given functional, performance, and specific quality requirements." [Gli07]*

By showing multiple representation forms of the same requirement, Glinz [Gli07] further says: "[...] the kind of requirement depends on the way we represent it." At MBC development, a functional requirement describes a behavior that can either be experienced by the customer or else is testable. From observation it can be said, that all such requirements can be represented in the data formats used in later formalization steps. The focus of further discussion will therefore be on all requirements that can be represented in language patterns used later on. This sets the scope of this work to all functional requirements and a selected set of other requirements. This selected set is exactly the set of requirements that is not functional but can also be represented in the language patterns. In addition, there is a set of non-functional requirements that constraint the solution space but cannot be represented in language patterns. These requirements can be processed directly to the state machine without language pattern processing. Thus, the term operationalization of functional constraints shall be defined:

Definition 2.7. ***Operationalization of Functional Constraints****: "The process to manipulate functional constraints from its initial equation representation to be directly included in the state machine representation."*

To ensure a high quality system specification, requirements should be based on the following principles of the Institute of Electrical and Electronics Engineers (IEEE) and International Organization for Standardization (ISO). Specifically, the IEEE norm [CB98] and ISO-29148 [ISO11] provide definitions under "characteristics of individual requirements":

Definition 2.8. ***Characteristics of individual requirements***

(2.8.1) Necessary: The requirement defines an essential capability, characteristic, constraint, and/or quality factor.[...]

(2.8.2) Implementation Free: Objective is to be implementation-independent.

(2.8.3) Unambiguous: The requirement is stated in such a way so that it can be interpreted in only one way.[...]

(2.8.4) Consistent: The requirement is free of conflicts with other requirements.

(2.8.5) Complete: The stated requirement needs no further amplification because it is measurable and sufficiently describes the capability and characteristics to meet the stakeholder's need.

(2.8.6) Singular: The requirement statement includes only one requirement [...]

(2.8.7) Feasible: The requirement is technically achievable, [...]

(2.8.8) Traceable: The requirement is upwards traceable to specific documented stakeholder statement(s) of need, higher tier requirement, or other source (e.g., a trade or design study). The requirement is also downwards traceable to the specific requirements in the lower tier requirements specification or other system definition artifacts. [...]

(2.8.9) Verifiable: The requirement has the means to prove that the system satisfies the specified requirement. Evidence may be collected that proves that the system can satisfy the specified requirement. Verifiability is enhanced when the requirement is measurable.

The given specification guidelines enable specification engineers to identify and document requirements for any given system. The actual representation form is not defined. Section 2.3 discusses a variety of specification representation forms in the context of knowledge representation.

2.2.2. Testing - Verification and Validation

Testing is concerned with assessing product quality during system development. The International Software Testing Qualifications Board (ISTQB) and the Institute of Electrical and Electronics Engineers (IEEE) differ in their definition focus.

Definition 2.9. ***Testing[ISTQB15]*** *"The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects."*

Definition 2.10. ***Testing[ISTQB15](1)*** *"The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component."*

Definition 2.11. ***Testing[ISTQB15](2)*** *"The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items."*

ISTQB specifically states that scope of testing includes "all life cycle activities". It contains a list of specific tasks to be performed on the test object. This is not included in either IEEE definitions. All three definitions share two common aspects:

Commonality 1. *Specified behavior serves the purpose it is designed for*

Commonality 2. *Specified behavior is compared to observed behavior*

This work will use Definition 2.11 of [IEE90] as its working definition. It shall be mentioned that this definition is generalized beyond the scope of software items towards system items. Thus, testing shall be defined here as:

Definition 2.12. ***Testing (in this work)****: "The process of analyzing a* ***system*** *item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the* ***system*** *items."*

In this work, a feature is considered to be a functionality provided by the system. The observation throughout the comparison of the given definitions for testing revealed that all share the common goals of Commonality 1: Specification matches design purpose and Commonality 2: Specification matches observed behavior. Commonality 1 and Commonality 2 are both reflected in ISO-9000 [ISO05] definitions of validation and verification.

Definition 2.13. ***Validation [ISO05]****: "Confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled."*

Definition 2.14. ***Verification [ISO05]****: "Confirmation, through the provision of objective evidence, that specified requirements have been fulfilled."*

A more practical way of describing validation and verification is: "Did we design the right product?" (Verification) and "Did we design the product right?" (Validation). Validation takes a product-centered view from the customer perspective. It assures that the requirements lead to the product that is useful to the customer. In contrast, verification is concerned with the specification process. It confirms whether on a certain specification level the generated (more detailed) specification is in alignment with the previous (more high level) specification. "This is independent of any intended use or purpose." [SL12] In contrast, this can be achieved by testing the product behavior against the defined (testable) requirements. In this work, the main focus is on verification rather than validation and therefore from now on *'verification'* will be used interchangeably with *'testing'*. This bottom-up development is illustrated in the V-model in Figure 1.1. It shows that verification efforts occur during all testing phases and levels of the integration. System integration includes all activities concerned with assembling and validating components, systems and the final product. Components (or the smallest stand-alone units in a product) are verified, afterwards merged into more complex structures and validated as systems or systems of systems.

The V-model in Figure 1.1 illustrates the three core levels (component, system and product) and the integration phases. In addition, the unit tests occur during development. In general, costs to detect and fix a failure increase exponentially during the integration. Thus, it can be said, that the saving potential during later validation phases is higher. It is favorable to reduce test loads of later phases, either by removing unnecessary tests or shifting test load to earlier phases. In this work, the focus is on identifying and removing redundant tests. A redundant test is a test that has an equal description to another test and therefore does not produce new or further test result information when executed. Shifting tests to other (lower) test phases is not addressed further here; This is discussed by Schwarzl and Herrmann [SH18]. Definitions in the context of testing shall be given:

Definition 2.15. ***Test Object [ISTQB15]****: "The component or system to be tested."*

For the component level test, all component interfaces are simulated and tests are performed only on each component in isolation. Similarly, system level tests are performed for each system in isolation while system interfaces are simulated.

Definition 2.16. ***Test environment [ISTQB15]****: "An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test."*

This work distinguishes five different test environments (or platforms) as used at MBC development.

1. Software-in-the-Loop (SIL)
2. Hardware-in-the-Loop (HIL)
3. Compoenent-HIL (C-HIL)
4. Functional Mock-Up (FMU)
5. Vehicle

Remark. *This list is selective, based on the purpose discussed and not complete*

Definition 2.17. ***Test goal [SL12], all translated**:*

1. *Finding failures as the general goal of testing*
2. *Finding specific failures through adjusted test cases*
3. *Proof of requirement fulfillment through the test object as the specific goal of one or more test cases.*

Remark. *In this work, test goal shall be used as: "Proof of requirement fulfillment through execution without failure detection of all linked test cases, whereas each requirement must have at least one linked test case."*

The test object (see Definition 2.15) depends on the current integration phase and describes the object under test as well as the parts excluded from validation. Tests are performed in a test environment (see Definition 2.16). The test environment is selected based on the integration phase, test object and test goal (see Definition 2.17). Test environments are mentioned in this context to strengthen the argument that later test phases are more costly. The degree of manual work required for a (testing) task and the increased efforts for adjustment of the finding are the main drivers for costs. Software unit tests are (mostly) automated, C-HIL tests require a tester for setup and critical moments while vehicle tests are performed fully by a tester. The scope of a test is determined by the test goal. Spillner [SL12] distinguishes four kinds of testing:

Definition 2.18. ***Functional Testing[ISTQB15]**: "Testing based on an analysis of the specification of the functionality of a component or system."*

Definition 2.19. ***Non-functional Testing[ISTQB15]****: "Testing the attributes of a component or system that do not relate to functionality, e.g., reliability, efficiency, usability, maintainability and portability."*

Definition 2.20. ***Structural Testing[ISTQB15]****: "Testing based on an analysis of the internal structure of the component or system." (Definition for white-box testing)*

Definition 2.21. ***Regression Testing [ISTQB15]****: "Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. It is performed when the software or its environment is changed."*

According to the definition used for the test goal (see Definition 2.17), this work focuses on functional tests. Change-related tests can include a selected set of tests for each of the three other categories (dependent on performed system changes). Tests are often repeated numerous times during product development. Thus, optimizing such tests contains an enormous savings potential. Tests are defined, structured and documented in the form of test set, test case and test step. The common definitions in standards and norms differ from the terminology used at [MB17].

Definition 2.22. ***Test Set [ISTQB15]****: "A set of several test cases for a component or system under test, [...]"*

Definition 2.23. ***Test Case [MB17]****: "[...]. [A test case] is the aggregation of mutual test steps. The test case result is the aggregation of its test step results."*

Both definitions contain the idea that the defined term is a set of elements. The elements provide actual system tests. In this work, the term test case shall be used as defined at MBC in Definition 2.23. The meaning of test case is closely related to the test set Definition 2.22.

Definition 2.24. ***Test Case [IEE90]****: "A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement."*

Definition 2.25. ***Test Step [MB17]****: "A test step consists of a precondition, trigger and postcondition [...]. A test step proves (pass) or disproves (fail) a transition from one state (precondition) to another state (postcondition) caused by a specific trigger. Multiple test steps aggregate to a test case."*

The definitions for test case [IEE90] and test step [MB17] both contain precondition, postcondition, input values (triggers) with the goal of verification of requirements (precondition, trigger and postcondition). The term 'test step' shall be used as defined in Definition 2.25. It is closely related to the definition of test case at [IEE90]. Further the terms 'precondition', 'trigger' (state 'transition') and 'postcondition' shall be defined.

Definition 2.26. ***Precondition [ISTQB15]****: "Environmental and state conditions that must be fulfilled before the component or system can be executed with a particular test or test procedure."*

Definition 2.27. ***State transition [ISTQB15]****: "A variable (whether stored within a component [or system] or outside) that is read by a component [or system]."*

Definition 2.28. ***Postcondition [ISTQB15]****: "Environmental and state conditions that must be fulfilled after the execution of a test or test procedure."*

In this work, precondition and postcondition are seen as system states (as in a state machine). To account for output verification, either the expected result must contain a description of the outputs or postcondition includes output as one part of its description. This second option might not be the most elegant form of representation but it makes it possible to express test cases and test steps in the structure used in this work. The combination of current state and input determines a transition path and transition time (moment or period) where the system changes from the initial state (precondition) to the final state (postcondition). In this work, 'transition' is used synonymously with 'trigger'. The comparison of expected to actual results is performed for the precondition, transition and postcondition. A test is passed when the test description for the initial state, transition path and final state match the observed states and path in the actual system at the relevant observation moment during test execution. Existing test specification guidelines enable test specification engineers to derive and document tests and links to requirements for any given test object. The actual representation for the test description form is not defined. This specifically affects the description of precondition, transition and postcondition. Section 2.3 discusses a variety of representation forms in the context of knowledge representation as well as the possibility of applying these representation forms to test descriptions.

2.3. Knowledge Representations in Systems Engineering

Sharma and Biswas [MT13] make a strong statement that in order to make progress in requirements engineering, knowledge representations must be understood first. That includes clarifying and defining *knowledge* and discussing the field of representation forms. Knowledge is closely related to data and information. The three terms, knowledge, data and information, are often used in overlapping or interchangeable ways. Closer analysis makes it obvious that all three terms are related but build on each other rather than have the similar or overlapping meanings.

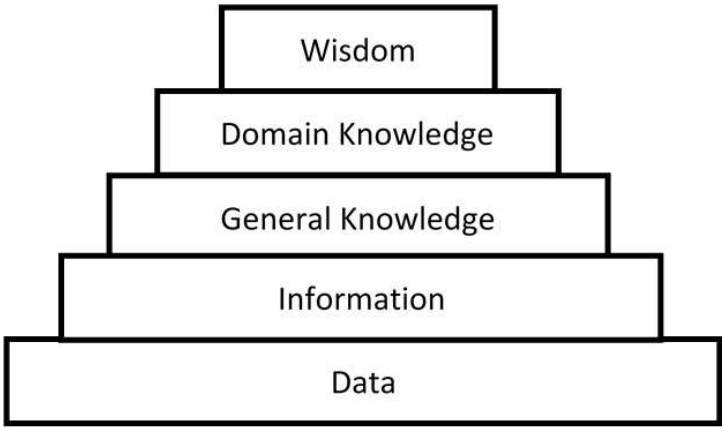


Figure 2.7.: Knowledge Representation: Knowledge Levels, [NK98], adapted

"Data represents the unstructured facts and figures, [...] At the next level information is structured data that is useful [...] in analyzing and resolving critical problems. [...] At the third level is general knowledge, which is obtained from experts based upon actual experience. While information is data about data, knowledge is basically information about information". [Thi99] Knowledge is either general (third level) or specific to a given domain (forth level). This work focuses in particular on (automotive) systems engineering knowledge (specifically requirements, development and test knowledge). The term domain knowledge is defined here as:

Definition 2.29. *"**Domain Knowledge** is the specific knowledge that exists about a particular field or discipline."*

Further discussion will in particular be about knowledge in the specific domains of requirement, development and testing. Data and all its evolving forms (information, knowledge and domain knowledge) require representation forms. The choice of representation

form is implicitly based on the predominant factor that is intended to be incorporated with this particular representation. This work shall discuss three attributes in this context in particular:

(1): Relations between elements

(2): Order of elements

(3): Representation form of statements

This is an individual choice but leads to the relevant representation forms for this work. Subsection 2.3.1 and Subsection 2.3.2 discuss knowledge representation in the form of unstructured and structured texts. Subsection 2.3.4 contains FOL and Linear Temporal Logic (LTL) operators. Here logic operators are used to structure expressions in terms of order of elements. Subsection 2.3.3 describes relations between elements through FSM.

2.3.1. Natural Language

The most intuitive and most used representation form in this context is text or, in other words, natural language. The goal of a knowledge representation form is mainly to represent a certain meaning. In language representations and logic, this is referred to as semantics.

Definition 2.30. ***Semantic [EO117]****: "Relating to meaning in language or logic."*

Expressiveness of a given semantic varies over representation forms. Allowing transformations from natural language to any other representation form, all information represented in natural language must be expressible in the other representation form. This characteristic shall be called expressiveness. The maximum expressiveness of natural language is the minimum required expressiveness for any other representation form, that natural language is transformed to. It is possible that a representation form has a higher expressiveness (e.g. formula). This applies if in a practical approach all knowledge is represented in natural language. In case other representation forms serve as input, in analogy to requirements about natural language expressiveness, all following representation forms must at least provide the expressiveness level of the alternative input representation form. Any semantic requires a set of rules. This influences expressiveness. In language and computer science, such a rule set is referred to as syntax. The English Oxford Dictionary [EO117] provides two definitions for syntax.

Definition 2.31. ***Syntax (1) [EO117]****: "The arrangement of words and phrases to create well-formed sentences in a language."*

Definition 2.32. ***Syntax (2) [EO117]****: "The structure of statements in a computer language."*

Natural language syntax overall is irregular, contains exceptions and is exposed to change and adaptations over time. Its major purpose is human readability, also called naturalness. Machine code decreases in readability (and naturalness) through an adjusted syntax. This adapted syntax generally enables machines to reason from knowledge.
"There is often a tension between naturalness and suitability for inference." [Cla96]
This is shown in Figure 2.8: Change from informal to formal representation leads to an increase in machine readability and reasoning capability while it decreases human readability.

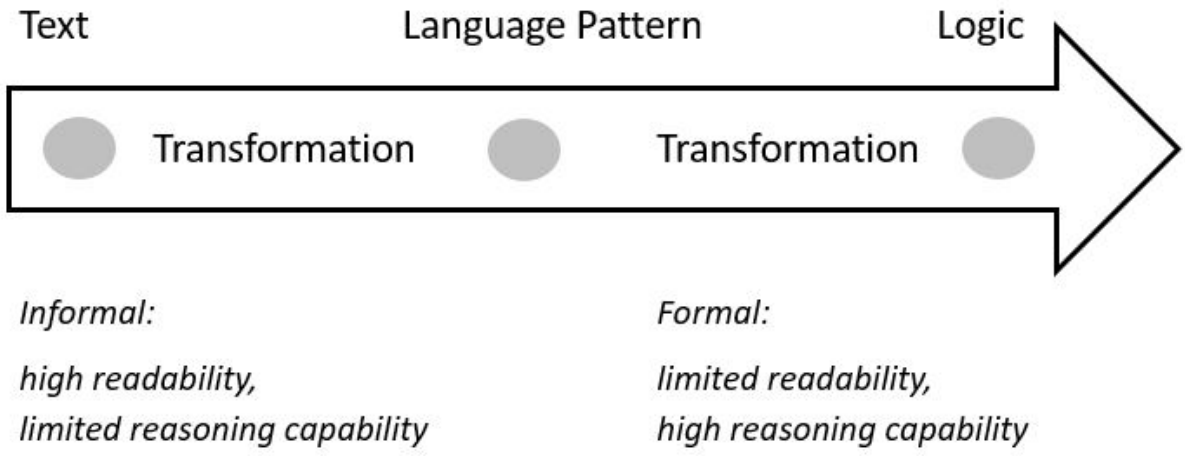


Figure 2.8.: Knowledge Representation: Informal and Formal

"Inference is the act of drawing conclusions about something on the basis of information that you already have." [Col17] The purpose of an inference engine is to increase the knowledge (about a system) by using the information (knowledge base) available. This means an inference engine can in principal draw conclusions about requirements (and tests) in a system and determine whether they are correct, complete and consistent. In a practical approach, the knowledge base can draw two kinds of information:

(1) Model structural information about the system (e.g. relations, links, order). This information is represented formally, thus in machine readable form and therefore generally suitable for inference without adjustment.

(2) Content information about the system (e.g. maximum car velocity). This information is represented informally and therefore not suitable for inference. The process to adapt such information is referred to as formalization.

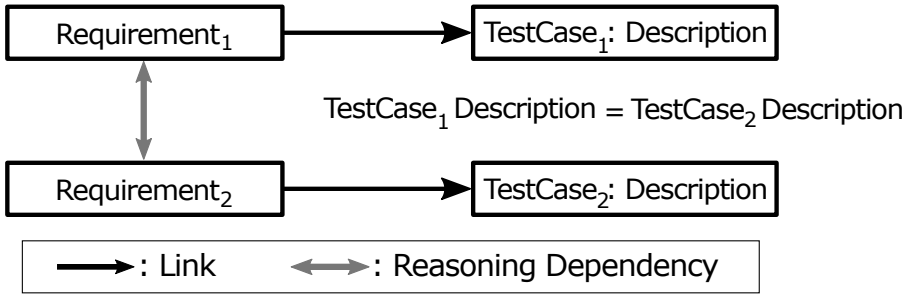


Figure 2.9.: Knowledge Transfer: Inference (Example)

The combination of structural and content-related information can be powerful for inference. It is assumed that $Requirement_1$ and $TestCase_1$ are linked and $Requirement_2$ and $TestCase_2$ are linked while no other links exist. In addition, it is assumed that the content of $TestCase_1$ and $TestCase_2$ is exactly identical (the most simple case for inference of informational data). Through inference it can be concluded that $Requirement_1$ and $Requirement_2$ are dependent without looking at the content of $Requirement_1$ and $Requirement_2$. It is necessary to convert the knowledge base from a natural representation form to a machine-readable form in order to enable such inference for more complicated cases than comparison for two representations containing exactly identical information. This process is referred to as formalization (see Section 3.3). An intermediate step from natural language to a final form that is fully suitable for inference is language patterns.

2.3.2. Language Patterns

Alexander described the abstract form of a pattern in the following way: *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."* [Ale85] Language patterns are becoming more common in requirements engineering. As Alexander said: patterns are a way to reuse the knowledge about found solutions while maintaining the capability to adjust for variations in the problem. Patterns provide a solution to a problem occurring in a transition from informal to formal requirements engineering.

Despite attempts in which requirements were documented solely in the form of models, graphs or other *higher* forms of representation, the state of the art automotive industry is still natural language text. This is driven by the fact that most projects include non-technical people (managers, stakeholders, lawyers). In addition, legal representation alone requires textual requirements. While general formalization could be achieved through various forms of representations like data bases, ontologies or semantic networks, this work (for the listed reasons about requirements) is limited to representations that can be derived from natural language. This can include the above listed representation forms, as long as a mapping or transformation from natural language exists. However, this work will now focus on representation forms that hold the potential to serve as such a mapping or transformation carrier.

Representation of requirements in natural language leads to a number of problems. While it is usually easier to write (and read) such requirements at first, textual descriptions are rather impractical in scaled form and for more complex situations. Subsection 2.2.1 listed the nine characteristics good requirements should contain according to ISO-29148 [ISO11]. The majority of these characteristics are hard to follow when requirements are textually stated. Mavin et al. [MWHN09] provide a list of possible problems: *ambiguity, vagueness, complexity, omission, duplication, wordiness, inappropriate implementation, untestability.* There exist many approaches that discuss ways to write natural language requirements while avoiding or limiting the listed problems. Wiegers [Wie99] discusses six of the nine characteristics and provides guidance about improving requirements quality. Similar, Hooks [Hoo94] describes characteristics of good requirements, common problems and provides advice, with practical examples on how to write better requirements. His work deviates further from the stated ISO-29148 [ISO11] but includes practical details (for example a discussion about critical words (e.g. *shall*), similar to the discussion in Master [Gmb16].)

The Volere Template by Robertson [RR00] focuses on a much more extended approach for writing requirements. It includes elicitation of requirements, different forms of project-related information representation (e.g. use cases, scenarios) and describes in detail an abstract method of representing functional and non-functional requirements. Lauesen [Lau08] covers a similar scope in his guide to requirements. It includes elicitation, considerations about testing as well as a limited discussion of requirements representation. The requirements guideline [MB07] and test specification guideline [MB17] can be seen

as tutorials or workbooks that acts as a guide through the whole process of handling requirements. However, neither document focuses on representation forms in particular. The publications by Wiegers [Wie99], Hooks [Hoo94], Robertson [RR00] and Lauesen [Lau08] are examples of qualitative advice in an abstract form. They all improve the specification process to a certain degree but since all four are still using unstructured text, the underlying problem is not addressed. Mavin states in [Mav12] *"Herein lies the dilemma: human beings want to write requirements in natural language, but it's too imprecise for the task."* He further concludes: *"In my view, the solution is a gently constrained application of English language."* Constrained language can be developed in form of templates. This work discusses a number of approaches. Yet the listed templates should be seen as examples, not as a complete representation of all existing work in the field of requirement specification templates.

(BIG) EARS - Mavin et al. [MWHN09, MW10, Mav12, MWGU16]
Introduction of EARS was based on the identification of common problems with requirements: ambiguity, vagueness, complexity, omission, duplication, wordiness, inappropriate implementation, untestability; (this list has already been stated above). EARS includes five syntax structures with examples (full description of each structure and examples in Mavin et al. [MWHN09]):

> **Ubiquitous**: The <system name> shall <system response>
> Example: *The control system shall indicate the engine oil quality to the aircraft*
>
> **Event-Driven**: WHEN <optional precondition> <trigger> the <system name> shall <system response>
>
> **Unwanted Behavior**: IF <optional precondition> <trigger>, THEN the <system name> shall <system response>
>
> **State-Driven**: WHILE <in a specific state> the <system name> shall <system response>
>
> **Optional Feature**: WHERE <feature is included> the <system name> shall <system response>

EARS concludes that the template addresses ambiguity, omission, duplication, wordiness and inappropriate implementation but requirements specified in EARS still contain problems in terms of untestability, complexity and vagueness.

Master - Sophist [Gmb16]
The intention for Master comes from the basic idea to "improve the quality of natural language requirements." Its goal is to improve the cost-to-value ratio for requirements specifications while maintaining a naturalness of language in the way requirements are stated. Master is based on five rules: Active phrasing (no passive requirements), complete sentences, main verbs, one process word entails exactly one requirement and the degree of detailing supports analysis.

Master distinguishes syntax structures for functional requirements, non-functional requirements and conditions. Non-functional requirements can be separated into quality, technical, user interface, additional services, tasks and legal requirements. This leads to three templates for non-functional requirements: property (all six), environment (technical) and process (tasks, legal). Condition specification is distinguished into logic, action and time-related specifications. Conditions are used as an addition to the other syntax structures.

> **Functional**: <opt: condition> <System> (<shall> or <should> or <will>) (<provide <actor> with the ability to> or <be able to>) <process verb> <object>)
> Example: *The system should provide the user with the ability to search items*
>
> **Non-functional - Property**: <optional condition> <characteristics> <subject matter> (<shall> or <should> or <will>) <be> <opt.: qualifying expression> <value> <system name> shall <system response>
>
> **Non-functional - Technical**: <(opt: component of the) subject matter> (<shall> or <should> or <will>) <be designed in a way> <opt: condition> <object> <can be operated> <characteristic> <(opt: qualifying expression)> <value>
>
> **Non-functional - Process**: <optional condition> <actor> (<shall> or <should> or <will>) <process verb> <opt.: details on process verb> <object> <opt: detailed definition of object>
>
> **Condition**: ((<if> <logical expression>) or (<as soon as> <event>) or (<as long as> <time period>))<requirements main clause>
> (Conditions preclude any of the others requirements templates)

The Master template provides variations for each structure. Thus, the real multiplicity for template patterns is much higher than the five initially given. It puts a special focus on main verbs. EARS uses only shall, while Master distinguishes shall, should and will.

Specification Pattern Systems (SPS) - Dwyer et al. [DAC98, DAC99, DP117]
The development of Specification Pattern Systems (SPS) is driven by the need for formal verification of requirements (e.g. model checking). A feasible approach therefore is temporal logic. SPS includes many forms of logic. This work only considers the patterns stated in LTL. Dwyer et al. argued that specification in LTL *"requires knowledge of several structured LTL idioms."* [DAC98] The templates therefore provide a natural language expression and a mapping to LTL for each structure. SPS contains eight patterns (two have multiple variants, so the total multiplicity is eleven). Each pattern is applicable in five cases: *'global', 'before R', 'after Q', 'between R and Q' and 'after Q until R';*

Universality: P is true
Example: *P is true before R ('global' pattern with 'before R' case)*
Server[off] is true before ElectricPowerSupply[on]

Absence: P is false

Existence: P becomes true

Bounded Existence: P becomes true
(P state occurs n-times)

Precedence: S precedes P

Precedence Chain: S, T precedes P
(2 causes 1 effect)

Precedence Chain: P precedes S, T
(1 cause 2 effects)

Response: S responds to P

Response Chain: P responds to S, T
(2 stimulus 1 response)

Response Chain: S, T respond to P
(1 stimulus 2 responses)

Constrained Chain Patterns: S, T without Z responds
(2-1 response chain with single proposition constraint)

The variations for SPS and with that its multiplicity are much wider than Master. The focus in all patterns is on expressiveness in LTL. This limits the natural expressiveness to a certain degree and compromises the implementation.

Overall, EARS and Master, based on the historic reason for invention, are both almost natural language implementations. SPS differs in so far, that it centers around the idea of formalization. Figure 2.8 shows that formalizing a representation improves the reasoning capability. By limiting the specification to language patterns, the syntax is in fact constrained in comparison to free text. This provides structure with the cost of reduced creativity and expressiveness. Two kinds of patterns can be observed. Domain independent templates and single domain templates. A domain-specific template tends to be easier to use in a specific problem setting, which is why specification engineers tend to prefer domain-specific templates. Due to the inherent domain limitation, the scope of single domain templates is limited (e.g. Li et al. [LNHK11] and Shen et al. [SPZ12]).

General approaches are harder to fit to a given problem and implementation is less intuitive, thus less favored by specification engineers. Yet domain independent approaches by design provide a much broader scope. In addition, the problem of inconsistencies between domain-specific models (or requirements) can be treated (see e.g. Helmig et al. [HKS+10], Berenbach and Wolf [BW07]). This represents a trade-off with no objectively superior solution but rather a subjective choice of preference. This work focuses on a wide applicability with the downside of a more inconvenient implementation. All three approaches shown (SPS, EARS and Master) fulfill this constraint. After selection of a multi-domain approach, this work is limited further: It continues to use only SPS by Dwyer [DAC98, DAC99]. In contrast to EARS and Master, SPS provides an empirical researched mapping to LTL. The advantages and implications are discussed in more detail in Section 3.3).

The initial intention for EARS and Master lies in the reduction or avoidance of common problems during the requirements specification process. Dwyer mentions that verification was one of the main drivers for developing SPS. Verification can be seen as a form of inference in this case. In the discussion about knowledge inference in Section 2.3.1, Figure 2.8 showed that a more formal representation is more suitable for inference. Using a template with predefined syntax structure allows for a much better inference. Thus, intended or not, all templates make it possible to reason about requirements through their syntactically restricted description. This reasoning capability shall be improved through further formalization. Subsection 2.3.4 addresses this specifically for logic while Chapter 3 derives a generalized process. Subsection 2.3.2 discussed expressiveness of representation forms. In the same context it can be asked, whether a particular repre-

sentation form can express a certain form of (domain) knowledge. Specifically here the question is *"Can tests be represented in the form of SPS?"* Since tests are derived from requirements, this must be possible. Every test description is based on the content of the requirement. The required expressiveness for a test-representation thereby can only decrease in relation to the required expressiveness of a requirements-representation. This assumption was confirmed in the analysis performed later where *"no incident was found, where a test could not be represented in a SPS."* It can be said that the shown language patterns (especially the SPS used later by Dwyer et al. [DAC98, DAC99]) can be used to express tests in addition to requirements.

2.3.3. Finite State Machines

It is a necessity in systems engineering to represent complex systems through formal models. Historically, requirements engineering was a primarily text-based discipline. While textual requirements are still common, formal modeling is becoming increasingly common. As discussed in Section 2.1, system structure and architecture can be represented best by class diagrams and functional representation is shown through sequence diagrams. The third common form of system representation is concerned with system behavior. Automata and specifically finite state machines serve well and are used most often to model behavior in requirement engineering. Figure 2.10 shows the four forms of machines included in automata theory. Turing machines are the most complex form of automata. "Turing machines are capable of changing symbols and simulate computer execution and storage." [ST118] "Turing machines can simulate any computer algorithm, no matter how complicated it is." [CAM18]. "A linear bounded automaton is a nondeterministic Turing machine." [Ben18]

"A pushdown automata is a finite automation with a stack. A stack can contain any number of elements, but only the top element may be accessed." [VIR10]. The simplest form of automata are finite state machines. "Finite state machines are only able to compute primitive functions." [ST118]. The scope of this work is limited to finite state machines, therefore pushdown automata, linear bounded automata and Turing machines will not be discussed further. However it shall be noted, that automata theory is closely related to formal language theory. This connection between language and model representations is further examined in Chapter 3.

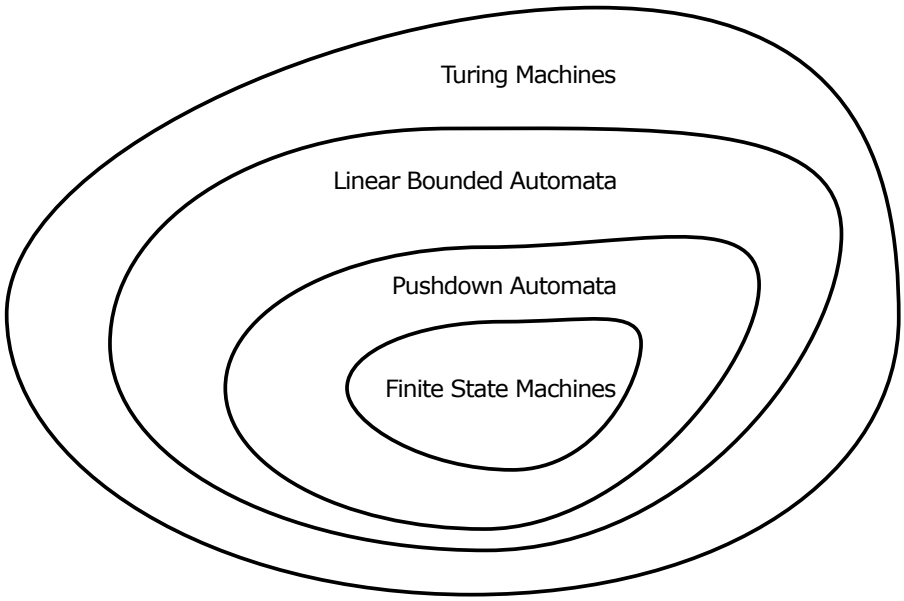


Figure 2.10.: Automata: Overview - Automata classes

A distinction is made between Finite State Machines (FSM) and infinite state machines (or transition systems). Infinite state machines contain no start and end state and the number of states and transitions can be infinite. In comparison to FSM, infinite state machines are more powerful. The characteristics of industrial systems (here specifically E/E Systems) usually contains a finite number of states and transitions. The characteristics of infinite states does not allow all conversions and analysis steps needed for the approach discussed in this work, which is why finite state machines are better suited for modeling systems in this context. We use the definition proposed by Cerny [Cer80] and used by Kam et al. [KVBSV12] and Villa et al. [VKBSV13] as their basis for FSM. The general definitions of Cerny [Cer80] include Definition 2.33 Characteristic Functions of S, Definition 2.34 Characteristic Function of R, Definition 2.35 Non-Deterministic FSM, Definition 2.36 State-Transition Graph, Definition 2.37 State Transition Relation T and Definition 2.38 for MooreDFSM.

Definition 2.33. ***Characteristic Function of*** S ***[KVBSV12, VKBSV13]:*** *Given a subset $S \subseteq U$ where U is some finite domain, Characteristic Function of $S, \chi_S : U \to B$, is defined as follows, for sequel $B = \{0, 1\}$. For each element $\chi \in U$,*

$$\chi_S(x) = \begin{cases} 0 & \text{if } x \notin S, \\ 1 & \text{if } x \in S \end{cases} \tag{1}$$

Definition 2.34 extends Definition 2.33 from one domain U to two domains X and Y.

Definition 2.34. ***Characteristic Function of*** R ***[KVBSV12, VKBSV13]:*** *Given a relation $R \subseteq X \times Y$, where X and Y are finite domains, Characteristic Function of $R\chi_R : X \times Y \to B$ is defined as follows. For each pair $(x, y) \in X \times Y$,*

$$\chi_R(x, y) = \begin{cases} 0 & \text{if } x \text{ and } y \text{ are not in relation } R, \\ 1 & \text{if } x \text{ and } y \text{ are in relation } R \end{cases} . \tag{2}$$

Definition 2.34 can be extended from two domains X and Y to n domains. Non-deterministic state machines are the most general form of FSM. The classification will distinguish the different forms of FSM and define the specific type that is used further in more detail.

Definition 2.35. ***Non-Deterministic FSM*** M_{ND} ***[KVBSV12, VKBSV13]*** *A non-deterministic FSM shall be defined as a 5-tuple where S represents the finite state space, I represents the finite input space and O represents the finite output space, T is the transition relation defined as a characteristic function $T = I \times S \times S \times O \to B$. On an input i, the FSM at a present state p transits to a next state n and output o if and only if $T(i, p, n, o) = 1$. There exists one or more transitions for each combination of present state p and input i. $R \subseteq S$ represents the set of reset states. Reset state for a state machine represents the starting state in which a state machine remains until the first input is given.*

$$M_{ND} = \{S, I, O, T, R\} \tag{3}$$

Definition 2.36. ***State-Transition Graph*** STG ***[KVBSV12, VKBSV13]*** *For a given FSM $M_{ND} = S, I, O, T, R$, the State-Transition Graph (STG) is defined as $(M) = \{V,E\}$, where each state $s \in S$ corresponds to a vertex in V labeled s and each transition $(i, p, n, o) \in T$ corresponds to a directed edge in E from vertex p to vertex n.*

$$STG(M) = \{V, E\} \tag{4}$$

Remark. *This work deviates from* ***Definition 2.27*** *in so far, that transitions are its own class of vertex objects, not simply directed edges. It is required that two state vertex elements are separated by a directed edge from the first state vertex to a transition vertex and another directed edge from the given transition to the second state vertex.*

Definition 2.37. ***State Transition Relation*** T ***[KVBSV12, VKBSV13]*** *State transition relation T must be complete with respect to i and p. This means there is*

always at least one transition out of each state on each input. A state transition $T = 1$ *is complete.*

$$\forall i \in I \ \forall p \in S \ \exists n \in S \ \exists o \in O \ \textit{such that} \ T(i, p, n, o) = 1 \tag{5}$$

The given definitions describe the most general form of a finite state machine: A non-deterministic finite state machine. This leads to the general inner structure of a finite state machine as shown in Figure 2.11.

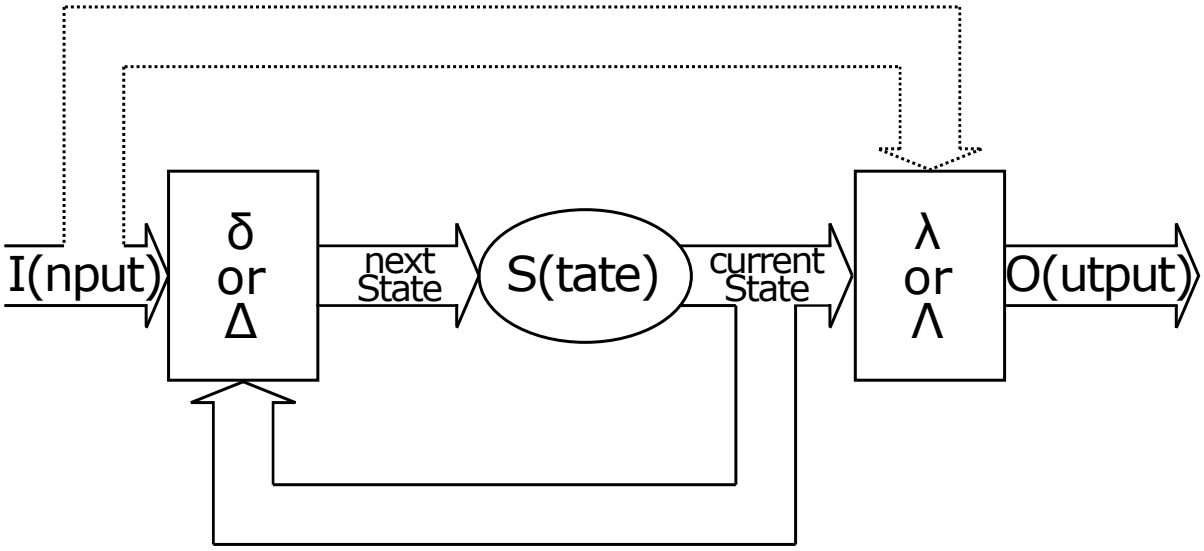


Figure 2.11.: Automata: Finite State Machine Structure, Martin [Mar18]

Input and current state both affect the transition function (δ (Moore) or Δ (Mealy)). This function leads to the next state and the output logic (λ (Moore) or Λ (Mealy)) that generates output o. Depending on the exact type of state machine, further limitations apply, mostly on the transition function. Figure 2.12 provides an overview over the different classes of finite state machines. Generally, FSM can be distinguished into non-deterministic (left branch) and deterministic (center branch) machines. Exceptions are Partially Non-Deterministic Finite State Machine (PNDFSM) and Incomplete Specified Finite State Machine (ICSFSM) (right branch). For PNDFSM, the next state is deterministic, but since the output is non-deterministic, it is overall pseudo-deterministic. ICSFSM is incompletely specified for both, the next state and the output, and thus it does not necessarily behave deterministically for a given next state or output. Inconsistent behavior does not suit the models used in this work, which is why this branch (right side) will not be discussed in more detail. Non-deterministic state machines are more

general than deterministic state machines. They contain a non-empty set of next states and outputs for a given combination of current state and input. Since the multiplicity is ≥ 1, it is not deterministic which next state or output is chosen.

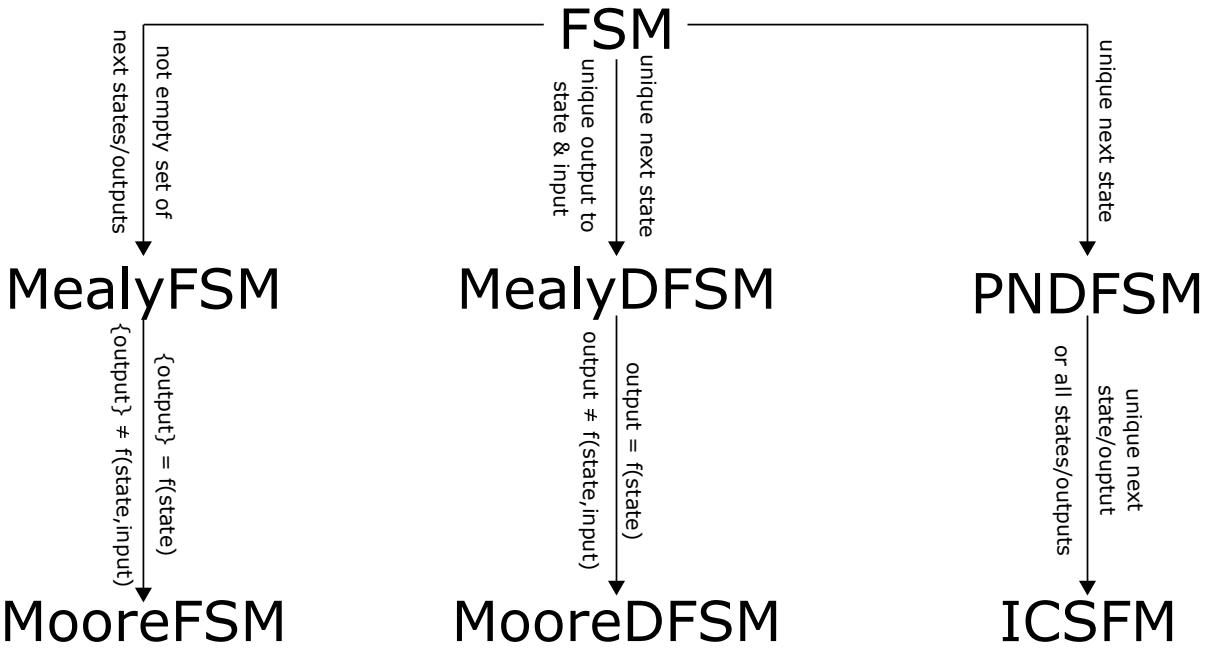


Figure 2.12.: Automata: State Machine Classification, Martin [Mar18]

Non-Deterministic states have transition function (δ or Δ) where each combination of input and current state is mapped to one or more unique next states. In addition, states have an output function (λ or Λ) where each combination of input and current state is mapped to one or more unique outputs.

Deterministic states have transition function (δ or Δ) where each combination of input and current state is mapped to a unique next state. In addition, states have an output function (λ or Λ) where each combination of input and current state is mapped to a unique output.

Besides determinism, finite state machines can be divided into Mealy Finite State Machines (Mealy FSM) and Moore Finite State Machine (Moore FSM). Mealy FSMs and Moore FSM can be distinguished in their output function (λ (Moore) or Λ (Mealy)). Output functions in Moore FSM only depend on the *current_state* while Mealy FSM output functions include *current_state* and *input*. Figure 2.11 shows the difference.

While Moore FSM is shown with continuous lines, Mealy FSM adds the dependency between input and output functions shown with the dashed lines.

Mealy Deterministic FSM (Mealy DFSM) $output(\lambda) = f(current\_state, input)$
Moore Deterministic FSM (Moore DFSM) $output(\Lambda) = f(current\_state)$

Automotive systems are designed to behave in a predictable, repeatable way. Non-deterministic models violate that principle. Therefore, only deterministic finite state machines are considered further. To decide whether Mealy DFSM or Moore DFSM are more suitable in this context, it has to be analyzed how *output* should be created. Considering the goal of using a state machine as a specification, it seems practical that a given *current_state* produces the same output every time, regardless of *input*. *Input* (e.g. through pressing a button), might not affect output. Therefore, all further models are performed on Moore DFSM. There are certainly situations and setups where a Mealy DFSM would be more suitable, yet in the majority of use cases for specifying a system, a Moore DFSM seems appropriate. Therefore, a Moore DFSM which depends only on *current_state* is chosen for this work.

Definition 2.38. ***Moore DFSM*** *M_{MD}[KVBSV12, VKBSV13] shall be defined as a 6-tuple $M_{MD} = \{S, I, O, \delta, \lambda, r\}$. In analogy to Definition 2.35, S represents the finite state space, I represents the finite input space and O represents the finite output space. δ is the next state function defined as $\delta : I \times S \to S$ where $n \in S$ is the next state of present state $p \in S$ on input $i \in I$ if and only if $n = \delta(i, p)$. λ is the output function defined as $\lambda : S \to O$ where $o \in O$ is the output of present state $p \in S$ if and only if $o = \lambda(p)$. $r \in S$ represents the reset state.*

$$M_{MD} = \{S, I, O, \delta, \lambda, r\} \tag{6}$$

Definition 2.39. ***Mealy DFSM*** *shall be defined in the same way a Moore DFSM was specified in Definition 2.38. The only adjustment is that for Mealy DFSM the output function is defined as $\Lambda : S \to O$ where $o \in O$ is the output of present state $p \in S$ and $i \in I$ if and only if $o(\Lambda) = (p, i)$.*

Based on the above definitions of Kam et al. [KVBSV12] and Villa et al. [VKBSV13], the general principles of FSM are shown and described. It is possible to categorize FSM as non-deterministic and deterministic as well as distinguishing between Moore

DFSM and Mealy DFSM. Specific, more detailed definitions for State, Input, Output, Transition Logic, Output Logic and reset state are needed for the Mealy DFSM used in the given context. Definitions for all six Mealy DFSM variables are provided below. This might deviate from the more detailed definitions in Kam et al. [KVBSV12] and Villa et al. [VKBSV13].

Definition 2.40. ***State*** *S shall be defined as a unique, valid and particular condition that a system can be in. It is described through the sum of all system attributes, where each attribute contains a particular value or value range. The combination of all possible attributes with all possible values describes the upper limit of possible states. It is possible, that in a given context not all theoretically existing states are correct and accepted system states. There exists a start_state, an end_state and possible resetstates (in our case reset state is equal to initial state). Current_state and next_state are elements of the set of accepted system states.*

Definition 2.41. ***Input*** *I contains a limited number of symbols which are elements of a predefined set of accepted symbols (input alphabet). Input and current_state are inputs to the transition logic. Input affects the system.*

Definition 2.42. ***Output*** *O contains a limited number of symbols which are elements of a predefined set of accepted symbols (output alphabet). Output is generated by the outputlogic and does not affect the system.*

Definition 2.43. ***Transition Logic*** *δ progresses the system from current_state to next_state. It uses current_state and input as incoming variables for the logic function to determine a unique next_state. Multiple unique transitions can reach the same next_state. Occurring transitions can be distinguished into spontaneous, event driven, counter driven and timer driven.*

(2.43.1) Spontaneous Transition: *Input* does not need to meet a particular conditions. Transition can occur instantaneously.

(2.43.2) Event Driven Transition: *Input* has to meet a given input condition. Fulfilling that conditions causes the transition. Counter and Event Driven Transitions are special forms of Event Drive Transitions.

(2.43.3) Counter Driven Transition: *Input* contains a counter which has to fulfill a given counter condition (number of repetitions where the counter is increased). Meeting the condition causes the transition and resets the counter.

(2.43.4) Timer Driven Transition: *Input* contains a timeout signal which causes the transition to wait until fulfilled (wait for a given time span). Once the timeout signal condition is met, transition takes place.

Definition 2.44. ***Output Logic*** *λ generates an output based on current_state. Outputlogic does not affect the system.*

Definition 2.45. ***Reset State*** *R retires the system to a predefined state (e.g. system shutdown or reaching an end_state). In this case reset_state is equal to initial_state.*

A system can be modeled through a *start_state* which transitions to a *next_state* based on *input*. *Next_state* becomes *current_state*. It triggers *outputlogic* which generates an *output*. New *input* or fulfillment of the relevant transition condition in combination with *current_state* leads to *next_state*. Thus, the system can change between states through different inputs and generate outputs. The described Mealy DFSM is used to generate system models in Chapter 3 and Chapter 4.

2.3.4. First Order and Temporal Logic

This subsection contains the definition of a formal system under consideration of FOL and temporal logic. The system is defined through states and transitions between these states. The full set of system states can be reduced through requirements that invalidate a number of system states. The remaining states represent the set of valid system states. This is a subset of the total set of system states. Similarly, transitions can be invalidated through requirements, which reduces the set of transitions to valid transitions. Requirements (from now on called *conditions*) are expressed through parameters and logic quantifiers. Parameters are equal to a particular value for a given state. Quantifiers (from now on called *operators*) consist of FOL operators ('combination' (AND) $\wedge$, 'alternative' (OR) $\vee$, 'negation' (NOT) $\neg$, 'implication' (IMP) $\longrightarrow$) as well as temporal logic operators ('next' $\circ$, 'future' $\diamond$, 'global' $\Box$, and 'until' $\mathcal{U}$,). The listed operators are subsets of the total set of operators in first order, respectively LTL. All FOL operators can be expressed through the given operators [BN13]. Temporal logic can be reduced to expressions with the restrictive 'until' operator $\mathcal{U}^*$[ST118]. Therefore, from now on, wherever 'until' is mentioned, it refers to the restrictive version. For simplification the restrictive version is represented as $\mathcal{U}$ instead of $\mathcal{U}^*$.

Definition 2.46. ***System*** $\mathcal{S}$
Let $\mathcal{S}$ *be a system with the state space* $\mathcal{X}$*, where* $\mathcal{X} = \{set\ of\ states\}$.

Remark. *System* $\mathcal{S}$ *is seen here as an abstract representation of a system which is not necessarily equal to the term system as used later in this work.*

Definition 2.47. ***Parameter*** p
Let p_i *be a parameter where* p_i*:* $\mathcal{X} \longrightarrow \mathcal{P}_i$ *and* $\mathcal{P}_i$ *includes all values of the value set that parameter* p_i *can take.* $i \in \mathcal{I}$*, where* $\mathcal{I} = N = \{1, \ldots, n\}$ *since* $n \in \mathcal{N}$.

Information about two parameters p_1 and p_2 at the same time can be represented through the cartesian product. The parameter $p_1 \times p_2$ is defined as:

Theorem 2.1. ***Cartesian product***
Let the cartesian product be defined as: $p_1 \times p_2 : \mathcal{X} \longrightarrow \mathcal{P}_{i_1}(x) \times \mathcal{P}_{i_2}(x)$*; for* $i \in \mathcal{I}$.

Remark. *Parameter p is seen here as an abstract representation of a parameter which is not necessarily equal to the term parameter as used later in this work.*

Definition 2.48. ***Condition*** $\mathcal{C}$ *Conditions are requirements that a parameter must obey. Conditions are subsets of the total value set of parameters. They translates to the request that parameter* p_1 *can only equal particular values* $v_1, \ldots, v_k$ *where* $v_1, \ldots, v_k$ *are values of* p_1*. Thus, let* $v_1, \ldots, v_k \in \mathcal{P}_1$*. The subset* $\mathcal{Q}_i \in \mathcal{P}_i$ *consists exactly of all values, that are valid values where* $p_i = v$ *and* $v \in \{v_1, \ldots, v_k\}$*. Let* $\mathcal{Q}_i$*:* $= \{v_1, \ldots, v_k\}$*. The request that* p_i *can take or equal any value from* $v_1, \ldots, v_k$ *translates to the demand for the state* $x \in \mathcal{X} : p_i \in \mathcal{Q}_i$.
Example:
Parameter p_i *can take whole-number values between 0 and 10.*
Definition: $\mathcal{Q}_{i_1}$*:* $= \{v \in \mathcal{P}_{i_1} |\ 0 \geq v \geq 10\}$
Demand $p_i(x) \in \mathcal{Q}_{i_1}$*, that means:* $0 \geq pi(x) \geq 10$*. Let* $p_i(x) \in \mathcal{Q}_i$*, for state* $x \in \mathcal{X}_i$

Remark. *Condition* $\mathcal{C}$ *is seen here as an abstract representation of a requirement that the abstract system* $\mathcal{S}$ *must fulfill. The condition is constraining the state space of a parameter to its valid values. A requirement as used later can specify the system within the remaining state space. The term requirement is therefore not equal to the term requirement as used later in this work. To separate the different meanings of requirement, requirements in the abstract context are called conditions.*

Multiple conditions for (different) parameters $p_1, \ldots, p_k$ with $i_1, \ldots, i_k \in \mathcal{I}$ and $k \in \mathcal{N}$ can be combined. This leads to set-theoretic operations on the cartesian product of the value set. The FOL operators ('combination' (AND) $\wedge$; 'alternative' (OR) $\vee$, 'negation' (NOT) $\neg$ and 'implication' (IMP) $\longrightarrow$) as well as temporal logic operators ('next' $\circ$, 'future' $\diamond$, 'global' $\Box$ and 'until' $\mathcal{U}$) will now be discussed one by one.

Definition 2.49. ***Operation 'combination' (AND)*** $\wedge$

The combination of conditions can be represented through the given value ranges of the given subsets $\mathcal{Q}_{i_1} \in \mathcal{P}_{i_1}, \mathcal{Q}_{i_2} \in \mathcal{P}_{i_2}$ *for parameter* $p_{i_1} : \mathcal{X} \longrightarrow \mathcal{P}_{i_1}$, $p_{i_2} : \mathcal{X} \longrightarrow \mathcal{P}_{i_2}$, *where both conditions must be fulfilled. Thus, for a condition* $(p_{i_1}(x) \in \mathcal{Q}_{i_1}) \wedge (p_{i_2}(x) \in \mathcal{Q}_{i_2})$ *this translates into the set-theoretic union of both value sets:* $\mathcal{Q}_{i_1} \times \mathcal{P}_{i_2}$ *and* $\mathcal{P}_{i_1} \times \mathcal{Q}_{i_2}$.
It is given that:
$(p_{i_1}(x)) \in \mathcal{Q}_{i_1}$
$\Longleftrightarrow (p_{i_1}(x),\, p_{i_2}(x)) \in \mathcal{Q}_{i_1} \times \mathcal{P}_{i_2}$
$\Longleftrightarrow (p_{i_1}(x),\, p_{i_2}(x)) \in \mathcal{P}_{i_1} \times \mathcal{Q}_{i_2}$
therefore the condition can be expressed in logic form and as a set-theoretic expression:
$(p_{i_1}(x) \in \mathcal{Q}_{i_1}) \wedge (p_{i_2}(x) \in \mathcal{Q}_{i_2})$
$\Longleftrightarrow (p_{i_1}(x),\, p_{i_2}(x)) \in \mathcal{Q}_{i_1} \times \mathcal{P}_{i_2} \wedge (p_{i_1}(x),\, p_{i_2}(x)) \in \mathcal{P}_{i_1} \times \mathcal{Q}_{i_2}$
$\Longleftrightarrow (p_{i_1}(x),\, p_{i_2}(x)) \in \mathcal{Q}_{i_1} \times \mathcal{Q}_{i_2} = \mathcal{Q}_{i_1} \times \mathcal{P}_{i_2} \cap \mathcal{P}_{i_1} \times \mathcal{Q}_{i_2}$

Definition 2.50. ***Operation 'alternative' (OR)*** $\vee$

The representation of two alternative conditions, where at least one must be fulfilled, can be represented through the value set of the subset $\mathcal{Q}_{i_1} \in \mathcal{P}_{i_1}$, $\mathcal{Q}_{i_2} \in \mathcal{P}_{i_2}$ *for parameter* $p_{i_1} : \mathcal{X} \longrightarrow \mathcal{P}_{i_1}$, $p_{i_2} : \mathcal{X} \longrightarrow \mathcal{P}_{i_2}$. *The condition therefore is that at least one of the demanded requirements is met, thus:*
$(p_{i_1}(x) \in \mathcal{Q}_{i_1} \vee p_{i_2}(x) \in \mathcal{Q}_{i_2}$
$\Longleftrightarrow (p_{i_1}(x),\, p_{i_2}(x)) \in \mathcal{Q}_{i_1} \times \mathcal{P}_{i_2} \vee (p_{i_1}(x),\, p_{i_2}(x)) \in \mathcal{P}_{i_1} \times \mathcal{Q}_{i_2}$
$\Longleftrightarrow (p_{i_1}(x),\, p_{i_2}(x)) \in \mathcal{Q}_{i_1} \times \mathcal{P}_{i_2} \cup \mathcal{P}_{i_1} \times \mathcal{Q}_{i_2}$

Definition 2.51. ***Operation 'negation' (NEG)*** $\neg$

Raising a condition based on the given value set through the subset $\mathcal{Q}_{i_1} \in \mathcal{P}_{i_1}$ *for a given parameter* $p_{i_1} : \mathcal{X} \longrightarrow \mathcal{P}_{i_1}$ *which shall* ***not*** *be fulfilled, can be represented in the form of the negated condition:* $\neg(p_{i_1}(x)) \in \mathcal{Q}_{i_1}$. *This translates to the set-theoretic complement of the subset* $\mathcal{Q}_{i_1} \in \mathcal{P}_{i_1}$. *It represents the negation of a singular condition against a state. For a given state* $x \in \mathcal{X}$, *the condition that parameter* p_i *equals values*

in $\mathcal{Q}_i \in \mathcal{P}_i$ is represented as $p_i(x) \in \mathcal{Q}_i$. If that condition is not fulfilled, it is represented as $\neg(p_i(x) \in \mathcal{Q}_i)$. This is equivalent to:
$p_i(x) \in \mathcal{Q}_i^c = \mathcal{P}_i \setminus \mathcal{Q}_i$
$= \{v \in \mathcal{P}_i | v \notin \mathcal{Q}_i\}$
$= \{v \in \mathcal{P}_i | \neg(v \in \mathcal{Q}_i)\}$
Thus, there exists $\neg(p_i(x) \in \mathcal{Q}_i) \Longleftrightarrow p_i(x) \in \mathcal{Q}_i^c$ for a given state $x \in \mathcal{X}$.

Definition 2.52. ***Operation 'implication' (IMP) $\longrightarrow$***
Let there be two conditions for a given state, where for each condition a parameter must equal a certain value. Now let one of the given conditions cause the other condition. This shall be represented through: $\_1 \Longleftrightarrow \_2$, where $\_1$ represents the first condition and $\_2$ the second. Let p_{i_1} be the first parameter with condition $p_{i_1}(x) \in \mathcal{Q}_{i_1}$ for state $x \in \mathcal{X}$ and let p_{i_2} be the second parameter with condition $p_{i_2}(x) \in \mathcal{Q}_{i_2}$ for state $x \in \mathcal{X}$.
Thus $(p_{i_1}(x) \in \mathcal{Q}_{i_1}) \longrightarrow (p_{i_2}(x) \in \mathcal{Q}_{i_2})$
$\Longleftrightarrow (p_{i_1}(x) \in \mathcal{Q}_{i_1} \Longrightarrow p_{i_2}(x) \in \mathcal{Q}_{i_2})$

Theorem 2.2. ***Transformation of implication into combination, alternative and negation***
'Implication' $\longrightarrow$ shall be expressed through the previously defined operators 'combination' $\wedge$, 'alternative' $\vee$ and 'negation' $\neg$.

Let $p_{i_1}(x) \in \mathcal{Q}_{i_1} \wedge p_{i_2}(x) \in \mathcal{Q}_{i_2}$, so this statement is true based on the given condition of $p_{i_1}(x) \in \mathcal{Q}_{i_1} \Longrightarrow p_{i_2}(x) \in \mathcal{Q}_{i_2}$ since $p_{i_1}(x) \in \mathcal{Q}_{i_1}$ leads to $p_{i_2}(x) \in \mathcal{Q}_{i_2}$.
Therefore it is given that:
$p_{i_1}(x) \in \mathcal{Q}_{i_1} \wedge p_{i_2}(x) \in \mathcal{Q}_{i_2}$
$\Longleftrightarrow (p_{i_1}(x),\, p_{i_2}(x)) \in \mathcal{Q}_{i_1} \times \mathcal{Q}_{i_2}$

Let $p_{i_1}(x) \in \mathcal{Q}_{i_1}^c \wedge p_{i_2}(x) \in \mathcal{Q}_{i_2}$, which is true based on $p_{i_1}(x) \in \mathcal{Q}_{i_1}^c$ being true. If the first condition is not fulfilled, there follows no implication about the second condition. Let the statement $p_{i_1}(x) \in \mathcal{Q}_{i_1}^c \wedge p_{i_2}(x) \in \mathcal{Q}_{i_2}^c$ be true, it is implied that statement $p_{i_1}(x) \in \mathcal{Q}_{i_1}^c \wedge p_{i_2}(x) \in \mathcal{P}_{i_2}$ is true as well.

Let $p_{i_2}(x) \in \mathcal{Q}_{i_2}^c$, so $p_{i_1}(x) \in \mathcal{Q}_{i_1}$ cannot be true, since that would imply that $(p_{i_1}(x) \in \mathcal{Q}_{i_1} \Longrightarrow p_{i_2}(x) \in \mathcal{Q}_{i_2})$ is true. If $p_{i_1}(x) \in \mathcal{Q}_{i_1}$ is true, it implies that $p_{i_2}(x) \in \mathcal{Q}_{i_2}$ is true. In addition it follows that $p_{i_2}(x) \in \mathcal{Q}_{i_2} \cap \mathcal{Q}_{i_2}^c = \emptyset$ ↯. Therefore, it must be true that

$p_{i_1}(x) \in \mathcal{Q}_{i_1}^c$ as well as $(p_{i_1}(x), p_{i_2}(x)) \in \mathcal{Q}_{i_1}^c \times \mathcal{Q}_{i_2}^c$. The correct statements shall be aggregated into one overall statement.

Let $(p_{i_1}(x) \in \mathcal{Q}_{i_1} \longrightarrow p_{i_2}(x) \in \mathcal{Q}_{i_2})$
$\iff (p_{i_1}(x), p_{i_2}(x)) \in (\mathcal{Q}_{i_1} \times \mathcal{Q}_{i_2}) \vee (\mathcal{Q}_{i_1} \times \mathcal{P}_{i_2}) \vee (\mathcal{Q}_{i_1}^c \times \mathcal{Q}_{i_2}^c) = (\mathcal{Q}_{i_1} \times \mathcal{Q}_{i_2}) \vee (\mathcal{Q}_{i_1}^c \times \mathcal{P}_{i_2})$

Theorem 2.3. ***Transitions between States***
The System $\mathcal{S}$ shall be described in a way that it allows for dynamic examination. That means that the system can transition from a given current state $x_1 \in \mathcal{X}$ to a new current state $x_2 \in \mathcal{X}$. It shall be considered which states $x \in \mathcal{X}$ are accepted in order to allow the transition from $x_1 \in \mathcal{X}$ to $x \in \mathcal{X}$. This transition between states shall be described with a graph.

Definition 2.53. ***Graph $\mathcal{G}$***
Graph G is defined as $\mathcal{G} = (\mathcal{N}, \mathcal{E}, f)$, a triple of
$\mathcal{N}$ = set of nodes;
$\mathcal{E}$ = set of edges;
$f : \mathcal{E} \longrightarrow \mathcal{N} \times \mathcal{N}$, thus f is called assignment function f. Through assignment function f, each edge e is assigned a start point $s(e)$ and an end point (termination point) $t(e)$, therefore $f(e) = (s(e), t(e))$ and $\mathcal{N} = \mathcal{X}$.
$\mathcal{E}$ is the abstract set of edges. It consists of all possible transitions between states.
$\mathcal{E} := \{$ set of all possible transitions between states of the system $\mathcal{S}\}$; The flexibility to transition from one given state $x_1 \in \mathcal{X}$ to a second given state $x_2 \in \mathcal{X}$ through different transitions $e_1, \ldots, e_k$ is shown in Figure 2.13.

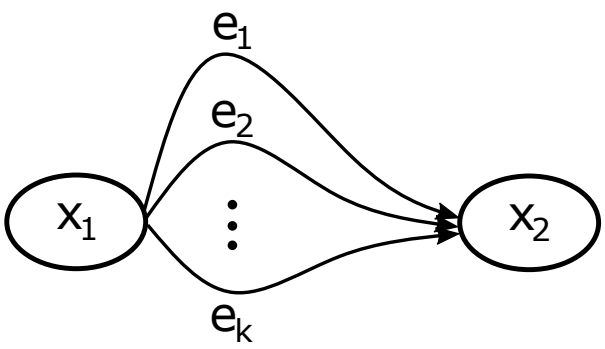


Figure 2.13.: Graph Representation: State-Transition Relation
(Transitions $e_1, \ldots, e_k$ between States x_1 and x_2)

Each particular transition e_i contains a specific description which characterizes the transition in detail. Assignment function f assigns a start state and end state to each transition:
$f : \mathcal{E} \longrightarrow \mathcal{N} \times \mathcal{N}$ and $e :\longrightarrow (s(e), t(e))$, where $s(e) \in \mathcal{N}$ represents the start state and $t(e) \in \mathcal{N}$ represents the end state. Shown in Figure 2.13 is that $s(e_i) = x_1$ and $t(e_1) = x_2$ for all transitions $e_1, \ldots, e_k \in \mathcal{E}$.

Theorem 2.4. ***States in time-dependent systems***
The system $\mathcal{S}$ can be in a current state $x \in \mathcal{X}$ at given point in time t_0. From this moment on, the system can change the current state from $x \in \mathcal{X}$ to $x' \in \mathcal{X}$ through any of the valid transitions. This is defined through Graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, f)$. Therefore function α is defined:
$\alpha : \mathcal{T} \longrightarrow \mathcal{N}$
$\alpha' : \mathcal{T} \longrightarrow \mathcal{G}$
$\mathcal{T}$ represents the set of points in time. For two consecutive points in time t_1, $t_2 \in \mathcal{T}$ with $t_1 \leq t_2$, for α the assignment is given that $(\alpha(t_1), \alpha(t_2)) \in \mathcal{E}$. This represents that the system transitions from current state $\alpha(t_1)$ to current state $\alpha(t_2)$. α' can be understood such a way that a given graph $\mathcal{G}$, as shown in Figure 2.14, has a valid state α' at any point in time $t \in \mathcal{T}$.

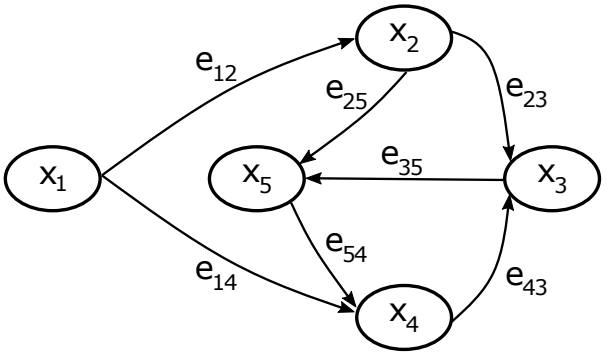


Figure 2.14.: Graph Representation: Time Dependent Systems

This relates to timed automata. Further discussion and classification is given by Waez et al. [WDR11]. The previously given definitions were FOL-related, and the second part of this section is dedicated to definitions and theorems for LTL. The motivation for temporal logic is founded in the need to describe time-related issues in a formal and mathematical form to represent time-based occurrences in an unambiguous way. The foundation

and first description for temporal logic was provided by Prior [Pri67, Pri03, PH03]. It contained four initial temporal operators (see [GG16]).

'past - eventually' - It has at some time been the case that...

'future - eventually' $\diamond$ - It will at some point be the case that...

'global past' - It has always been the case that...

'global future' $\Box$ - It will always be the case that...

Remark. *The focus is only on forward-directed operators, therefore 'past eventually' and 'global past' will not be considered and discussed further.*

One extension to Prior's initial temporal logic operators are the operators 'until' and 'since', which allow conditions in time descriptions to be included.

$\psi\ U\ \phi$: ψ will be true until a time when ϕ is true

$\mu\ S\ \varphi$: μ has been true since a time when φ was true

The initial definition of 'until' and 'since' by Kamp [Kam68] is reflexive. It allows the possibility of describing a past or future event and include the current state in this description. The stricter version for 'until' and 'since' is called irreflexive.

$\psi\ U^*\ \phi$: ψ will be true until a time when ϕ is true *(where ϕ is in the future)*

$\mu\ S^*\ \varphi$: μ has been true since a time when φ was true *(where φ is in the past)*

In this irreflexive version, ϕ is strictly in the future for 'until'. Similarly, φ has to be strictly in the past for 'since'. Kamp [Kam68] proved that all temporal logic operators can be expressed and therefore reduced to U^* and S^*.

Remark. *In alignment with the previous statement, the 'since' operator will not be considered further in this work.*

In addition to the previously defined FOL operators, temporal operators shall be defined. This allows the definition of time-based conditions, where the condition represented in the current state, affects a future (or previous) current state. Two forms of orders exist which must be distinguished: sequence-based orders and time-based orders.

Theorem 2.5. ***Sequence-based orders***
For a precise definition of a sequence-based order, the term of an oriented or directed edge path shall be defined. Figure 2.15 shows a representation of an edge path. Edge path: Let graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, f)$ *be a directed graph. A sequence* $e_1, \ldots, e_k \in \mathcal{E}, \mathcal{K} \in \mathcal{N}$ *of edges* $e_1, \ldots, e_k$ *is called edge path of current state* $x_1 \in \mathcal{X}$ *to current* $x_2 \in \mathcal{X}$*. This is valid if* $s(e_1) = x_1$*,* $t(e_k) = x_2$ *and* $t(e_i) = s(e_{i+1})$ *where for all* $i = \{1 \geq i \geq k-1\}$*.* k *represents the length of the edge path.*

Remark. *Edge paths of length 1 represent exactly one edge.*

Figure 2.15.: Graph Representation: Sequence-Based Order - Edge Path

For the description of states and possible transitions between states, the edge path represents a sequence of transitions, where for two consecutive transitions the end state of the first transition and the start state of the second transition must be identical. A transition represents an atomic state change.

Theorem 2.6. ***Time-based orders***
Operators for time-based orders are 'next' $\circ$*, 'global'* $\Box$*, 'future'* $\diamond$ *and 'until'* $\mathcal{U}$*.*

Definition 2.54. ***Operator 'next'*** $\circ$ *precedes a condition that, for a given current start state, shall affect the state which is the next current state after a transition changed the current state. This means that all states that can be reached from a given current state through a transition must fulfill the given condition. For a given current state* $x \in \mathcal{X}$ *and all states* $x' \in \mathcal{X}$ *reachable through a transition, the parameter* p_i *is defined with the parameter value* $p_i(x') \in \mathcal{Q}_i$*.* $p_i(x')$ *describes the situation, that all next current states* x' *must fulfill the given condition. For a given state* $x \in X$*, let:*
$\circ(p_i(x)) \in Q_i)$ *be the condition*
$\Longleftrightarrow$ *for all* $x' \in \mathcal{X}$
so that a given edge $e \in \mathcal{E}$ *with* $s(e) = x$*,* $t(e) = x'$ *exists, thus* $p_i(x') \in Q_i$*.*

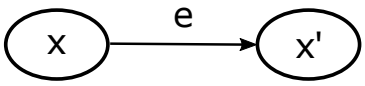


Figure 2.16.: Logic Representation: LTL Operator 'Next' $\circ$ (Example)

States $x' \in \mathcal{X}$ are possible next current states. They can be reached through transitions from the current state $x \in \mathcal{X}$.

Definition 2.55. ***Operator 'global'*** $\Box$

Operator 'global' $\Box$ *or 'for all' represents a condition for all states* $x' \in \mathcal{X}$ *that can be reached from an edge paths starting at the current state* $x \in \mathcal{X}$*. For all states* $x' \in \mathcal{X}$*, that can be reached through a sequence of transitions on graph* $\mathcal{G} = (\mathcal{N}, \mathcal{E}, f)$*, this condition applies and can be represented for a parameter* $p_i(x') \in \mathcal{Q}_i$ *through:*
$\Box(p_i(x)) \in \mathcal{Q}_i$
$\Longleftrightarrow$ *for all* $x' \in \mathcal{X}$
in the way that an edge path $e_1, \ldots, e_k \in \mathcal{E}$ *with* $s(e_1) = x$ *and* $t(e_k) = x'$ *exists for* $p_i(x') \in \mathcal{Q}_i$*.*

Figure 2.17.: Logic Representation: LTL Operator 'Global' $\Box$ (Example)

Theorem 2.7. ***Example LTL Operator 'Global'*** $\Box$

(2.7.1) Graph $\mathcal{G}$: Let $\mathcal{G} = (\mathcal{N}, \mathcal{E}, f)$ be a graph $\mathcal{G}$.

(2.7.2) Edge e: $\in \mathcal{E}$ is called cycle when start and end point of an edge are identical.

(2.7.3) Edge path $e_1, \ldots, e_k$: $\in \mathcal{E}$ is called circle when $s(e_1) = t(e_k)$

(2.7.4) Incoming valence of an edge x: $\in \mathcal{E}$ with $t(e) = x$

(2.7.5) Outgoing val. of an edge $x \in \mathcal{N}$ is number of edges $e \in \mathcal{E}$ with $s(e) = x$

(2.7.6) Valence of an edge $x \in \mathcal{N}$ is the sum of incoming and outgoing valences

Definition 2.56. ***Operator 'future'*** $\diamond$

Operator 'future' $\diamond$ *describes conditions against all possible infinity edge paths of transitions with a given start value from a state* $x \in \mathcal{X}$*. For a given edge* e_k *on an infinite*

edge path $e_1, e_2, \ldots \in \mathcal{E}$ *with* $s(e_1) = x$*, both start and end states of the edge must fulfill a condition. The condition is expressed through parameter* p_i *of* $t(e_k)$ *with values of* $\mathcal{Q}_i$*. This shall be represented as:*
$\diamond(p_i(x) \in \mathcal{Q}_i)$
$\Longleftrightarrow$ *for all infinite edge paths* $p = [e_1, e_2, \ldots]$ *in* $\mathcal{G}$ *there exists an edge* e_k *in the way that:* $p_i(t(e_k)) \in \mathcal{Q}_i$

Definition 2.57. ***Operator 'until'*** $\mathcal{U}$
Operator 'until' $\mathcal{U}$ *describes conditions against all edges paths* $p = [e_1, e_2, \ldots]$ *with a starting value from state* $x \in \mathcal{X}$*. All states on all infinite edges paths* $p = [e_1, e_2, \ldots]$ *must fulfill a condition until another condition is fulfilled. This means parameter* p_1 *shall have values in* $\mathcal{Q}_{i_1}$ *until parameter* p_2 *has values from* $\mathcal{Q}_{i_2}$*. It is expressed as:*
$\mathcal{U}(p_{i_1}(\_) \in \mathcal{Q}_{i_1},\ p_{i_2}(\_) \in \mathcal{Q}_{i_2})$
$\Longleftrightarrow (p_{i_1}(\_) \in \mathcal{Q}_{i_1})\ \mathcal{U}\ (p_{i_2}(\_) \in \mathcal{Q}_{i_1})$
$\Longleftrightarrow$ *(for all edge paths* $p = [e_1, e_2, \ldots]$ *with* $s(e_1) = x$*,* $p_{i_2}(t(e_k)) \in \mathcal{Q}_{i_2}$ *and* $p_{i_2}(s(e_i)) \in \mathcal{Q}_{i_2}^c$ *for all* $i = \{1, \ldots, k\}$*, let* $p_{i_1}(s(e_i)) \in \mathcal{Q}_{i_1}$ *for all* $i = 1, \ldots, k$*) and (for all edge paths* $p = [e_1, e_2, \ldots]$ *with* $s(e_1') = x$ *and* $p_{i_2}(s(e_i')) \in \mathcal{Q}_{i_2}^c$ *for all* $i = 1, 2, \ldots$ *let* $p_{i_1}(s(e_i')) \in \mathcal{Q}_{i_1}$*).*

The derived system $\mathcal{S}$ with graph $\mathcal{G}$, its states x and transitions, is the basis to apply the FOL operators 'combination' (AND) $\wedge$, 'alternative' (OR) $\vee$, 'negation' (NOT) $\neg$, 'implication' (IMP) $\longrightarrow$) as well as temporal logic operators ('next' $\circ$, 'future' $\diamond$, 'global' $\Box$, and 'until' $\mathcal{U}$ and describe conditions (requirements) against the system $\mathcal{S}$.

2.3.5. Deterministic versus Non-Deterministic Systems

In this chapter, different forms of (automotive) knowledge representations have been discussed. Motivation to look into more formal knowledge representations than natural language arises from the complexity increase in automotive system design. Fully described or specified cars should behave as a deterministic systems. All methods and processes are based on the assumption of an underlying deterministic system. Yet the rising inner complexity often prevents a full description of every system variable. In addition, informal natural language lacks the ability to express all aspects of the system. Both lead to an under-specification of the system. This makes the resulting system behavior appear to some extent non-deterministic when considering the existing specification.

Natural language can hardly describe all aspects of a complex multi-domain system. System descriptions are often incomplete. Once a system variable, system state or behavior is not fully described (*under-specified*), a certain degree of freedom for system behavior occurs. It is unclear how the system should behave in a situation of *under-specification*, thus it is non-deterministic in that sense. This phenomenon can be observed especially during system integration, particularly when two *under-specified* systems interact through their interfaces. Integration tests often reveal such non-deterministic behaviors. (e.g. tests at C-HIL or FMU show different system behavior than the same tests show when performed in a vehicle.) There are two solutions to the problem of systems that appear non-deterministic due to incomplete specification or under-specification. One is to change our intrinsic view and accept cars as non-deterministic systems. A consequence of this would be that all methods that build on the assumption of deterministic systems would no longer be suitable. The better solution is to replace (or at least extend) the incapable representation in the form of natural language by a more formal form. Such a form is complex and difficult, making it likely that specification efforts for requirement specification will increase. Through the representation of all relevant variables, a deterministic description of the system is regained. Formal representation adds to the often under-specified natural language specification and replaces this insufficient specification with a more complete specification.

Formal representations remove the degree of freedom that occurred in informal representation. Therefore, such a representation form requires the same or a higher expressiveness than natural language representations. Language patterns, state machines and logic are candidates for this. A feasible approach would be to represent systems in natural language and convert it step-by-step to the highest representation form. In each representation form, certain under-specified system elements become visible. Either the formal representation is adjusted or the informal form is extended and the result is mapped and reviewed. Such an approach requires mapping between the representation forms. This is discussed in detail in Chapter 3.

Remark. *Comment towards requirements and tests in terms of expressiveness: All approaches are performed with representations feasible for requirements. It is assumed based on empirical observation that tests generally have a lower expressiveness than requirements. Under that assumption, all representations that are suitable for requirements, are suitable for tests as well.*

2.3.6. Modeling Structures: Trees versus Graphs

In Subsection 2.3.1, the idea of separation between model structure and content information was shown. It seems feasible to initially treat both independently. Analyzing data structures shows two different problem classes that shall be defined:

Definition 2.58. ***Directed Cycle-Free Tree Structures (node valence $\leq$ 2):*** *"There exists a start and end node. The 'maximum node valence' for each node is two. Based on this, each node (except the start and end node) has exactly one predecessor and one successor. The nodes are connected, and the transitions between nodes provide an order that makes the tree a directed tree without any forks. The maximum node valence prevents the occurrence of cycles."*

Definition 2.59. ***Directed Cyclic Graph Structure****: "There exists a start node. Each node (except start and potential end nodes) has at least one predecessor and one successor. The graph can have forks, junctions and ring closures. There is not necessarily one particular end element. Transitions between nodes provide a direction. There can be multiple transitions (with multiple directions) between two nodes or transitions starting and ending on the same node. This shall be called a directed cyclic graph."*

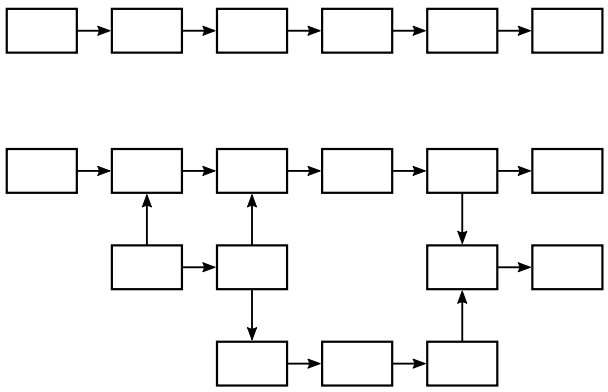


Figure 2.18.: Graph Representation: Directed Tree and Directed Cyclic Graph Structure

Any directed tree structures can be seen as a special form of a forward-directed cyclic graph structure. Inference combines model structure and information. Therefore, it is relevant what form of structure is present for a given problem or data set. Chapter 3 and Chapter 4 will use directed tree structures with one branch only (node valence ≤ 2) to formalize test cases. This represents a special solution. The generalization of the solution occurs when formalization for requirements is performed by using forward-directed cyclic

graphs. Directed tree structures without multiple branches have a clear order. Therefore, moving information between elements (e.g. 'next' $\circ$ operator in LTL) is rather simple. This becomes significantly more complex when forks, junctions and in particular ring closures occur. It therefore seems useful to distinguish upfront the two problem classes of directed tree structures and directed cyclic graph structures.

3. Novel Process Chain for Requirements and Test Formalization

> "The expression of a single requirement is a two-stage process, [...]. First, you have to determine the need and then you have to find a clear way to express it."
>
> Alistair Mavin [Mav12]

Requirements formalization is a wide field with a variety of approaches and solutions. Yet there is not one continuous method or process that has fully solved the formalization of automotive requirements and test data in a satisfying way. This chapter proposes a novel approach for formalization of requirements and test data from textual representation to a formalized model form. An overview of all consecutive steps is provided in Section 3.1. Section 3.2 discusses different representation forms during the requirements elicitation and documentation phases. There exist two feasible ways: *Classic* documentation in the form of Natural Language (NL) with subsequent conversion into SPS or direct specification in SPS. Both forms allow processing from pattern representation towards temporal logic form. This transformation is described in Section 3.3. Temporal logic expressions are compact representations of globally occurring logic dependencies. In order to achieve complete local representation, temporal logic has to be converted into locally representable logic. This can be achieved by using the underlying data structure. Section 3.4 discusses this mapping for directed one-branch trees, which are the simplest form of data structure for such problems. A generalization of this problem is mapping from temporal logic to FOL for directed cyclic graphs. This is covered in Section 3.5. To achieve an unambiguous representation, FOL can be sorted in normal form. Section 3.6 addresses this topic. The achieved formal and local representation of element descriptions allows for various applications as shown in Chapter 4.

3.1. Model Overview - Full Process Chain

The need for a reproducible and continuous formalization process was described above. The shown formalization approach focuses on functional requirements data and test descriptions for automotive systems, particularly in the electric/electronics domain. Overall, the process considers five representation forms with four transitions as shown in Figure 3.1. With the exception of the first transition (NL to SPS), all steps are automated. Automation requires a reproducible, consistent transformation rule set. Mapping rules for 'NL to SPS' as well as 'SPS to LTL' can only be validated empirically. Argumentation therefore lies in the informal domain space 'NL to SPS' (language space), and in the mapping between two inconsistent spaces (language space and logic space) for 'SPS to LTL'. Empirical validation is paired with case-based mapping rules for both steps. During all process steps, no distinction must be made between requirements and test data must be made, except for the third transition (LTL to FOL). In this transition, the underlying model data structure is considered. The structure differs for requirements and test data. The complex case for requirements therefore is treated with a case-based mapping while the special and more simple case of test data can be processed in a rule-based manner. The last transition derives CNF representations from FOL, which is a rule-based mathematical conversion.

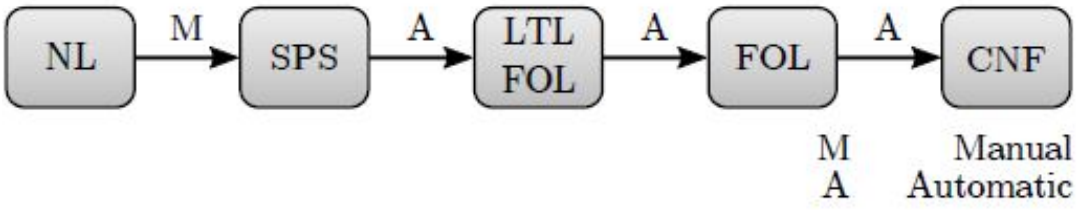


Figure 3.1.: Formalization Process Chain: NL to CNF, Walter et al. [WHPR17]

Overall, this approach achieves a conversion and formalization of data that is almost free of manual tasks. For a given set of requirements or tests as input with an initial representation in a structured textual form, this allows machine-based automated data optimization. This is important when considering scaled systems, particularly in industrial contexts. While such quantitative analyses are covered in Chapter 4, the detailed formalization approach is shown during the following sections of this chapter in a qualitative way. Each step is derived and explained in abstract form and performed with qualitative examples from industrial data from a case study of MBC systems. The lim-

itations for the formalization do not lie in the boundaries of the described system but in the expressiveness of all used representation forms. Specifically SPS was validated for functional requirements only, while it is not focused on any particular domain. Further discussion about the scope is provided in Section 3.2

3.2. Natural Language Expression Conversion to Specification Pattern Systems

In this section, the initial representation of requirement and test case descriptions including formalization towards specification patterns is addressed. There is no particular difference between requirements and test descriptions. For simplicity, all further explanations focus on requirements but methodically include test cases. Specification of requirements is mostly concerned with providing an accurate description of a given system condition or a functionality and proper documentation of that condition or functionality. Mavin [Mav12] expressed this as: *"The expression of a single requirement is a two-stage process, [...]. First, you have to determine the need and then you have to find a clear way to express it."* Subsection 2.2.1 listed the four core activities of requirements engineering as elicitation, documentation, verification & validation and management of requirements. These are illustrated in Figure 3.2.

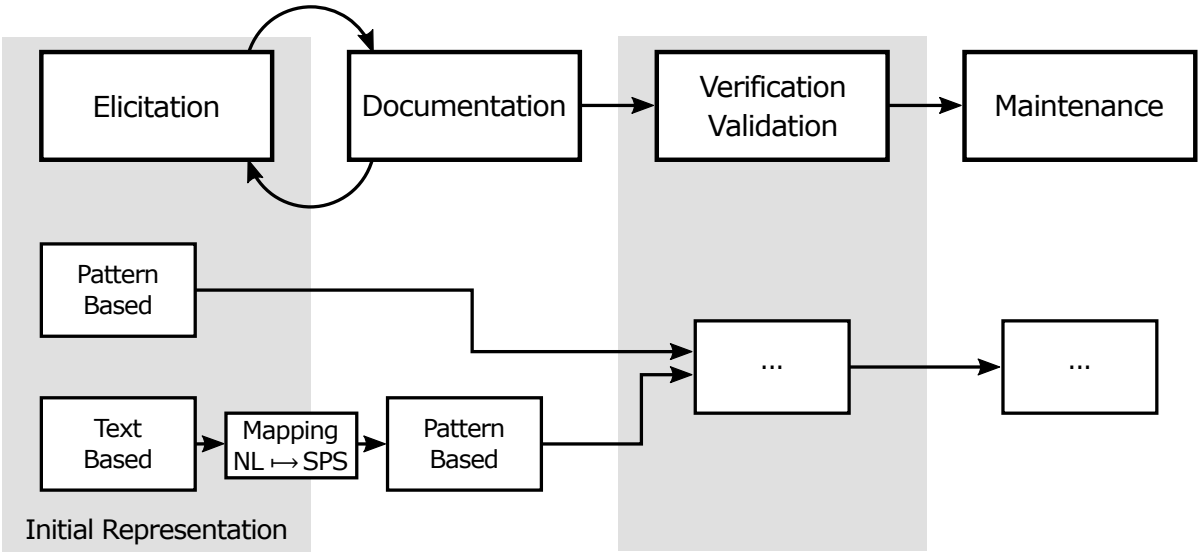


Figure 3.2.: Requirements Engineering: Elicitation and Documentation

Elicitation and documentation thereby occur in parallel in the form of a repetitive loop. While elicitation is concerned with finding or deriving the needed requirements for a system, documentation addresses the representation of these requirements. There are generally two forms of formalization feasible to this approach. These are the initial documentation in textual form and the manual conversion into patterns afterwards or the straight specification in patterns without the need for a mapping. While both approaches result in the same representation form, they occur in different moments during the specification process. Initial specification in natural language form takes place during each requirements elicitation and documentation cycle. It is separated from the mapping into specification patterns, which would naturally be performed after elicitation is completed. This approach allows finished specifications, that were initially not intended to be formalized, to be converted into specification pattern form. Similar to initial documentation in natural language, it is possible to specify requirements directly in specification patterns. Specification in patterns is potentially more complex than unrestricted specification, thus this might affect elicitation in a negative way. The advantage of straight pattern representation is that the mapping step is skipped, which reduces manual work effort. In contrast, some requirement, development and test engineers might prefer a natural language representation due to readability.

In Walter et al. [WHPR17], it was discussed, that no error introduction during manual formalization efforts (natural language to specification patterns) was observed. For the applications shown in Chapter 4, elicitation is already performed and a finished specification documentation exists. Conversion from natural language to specification patterns therefore occurs in a separate step after the elicitation. The general process of requirements elicitation will not be discussed in further detail here. It will be shown in Subsection 3.2.2 how initially textually represented requirements and tests can be converted into specification pattern form. Subsection 3.2.3 discuses how requirements can be directly documented in a specific set of specification patterns. Prior to these two approach variants, Dwyer's SPS are chosen as the specification patterns used in the context of this work. A more detailed assessment of SPS is performed in Subsection 3.2.1.

3.2.1. Selecting Dwyer's Specification Pattern Systems (SPS)

The advantages of language patterns were briefly discussed in Subsection 2.3.2. Language patterns assist the requirements documentation process by providing sentence structure,

limiting phrasings to particular words (e.g. verbs with legal bindings) and preventing common documentation errors like incomplete sentences. Selection of a particular language pattern for a given problem set must be based on the scope of the pattern with regards to the problem domain and the required expressiveness of its descriptions. Many language patterns are specified for a particular domain (e.g. static versus dynamic, software versus hardware, aerospace versus automotive, ...). Thus, a specific pattern might not contain the expressiveness for a given problem domain.

This work takes place in the automotive E/E domain. Subsection 2.3.2 listed Volere, Sophist Master, EARS and SPS as common language patterns. All four language patterns provide sufficient expressiveness to serve problems given in the automotive E/E domain. Additional factors must be considered for a systematic selection. The reason why SPS is chosen for this work is twofold. There exists previous work by Konrad and Cheng [KC02, KC05a, KC05b] as well as Post et al. [PMP11, PHP11, PH12, PMHP12] with SPS on embedded real-time automotive systems. While other industrial work using Volere, Sophist Master or EARS certainly exists, the cited work matches the problem domain of this work closely. This makes SPS the favorable choice. Second, the later processing steps in this work include mapping from language to logic space. It was discussed that such a mapping can only be based on empirical assignment between given language pattern expression and specific temporal logic expression. Dwyer et al. [DAC98, DAC99] provide such a mapping. This is discussed in more detail in Section 3.3. No other language pattern includes such a mapping.

This mapping is one of the essential building blocks of this work, which makes SPS the obvious language pattern of choice in this context. The intention to develop SPS as an additional form of language patterns is rooted in the idea to allow programmers a simple way to use finite state verification tools. Dwyer et al. [DAC17] argues that finite state verification requires a particular form of input. Most programmers (in regards to this work also requirements, development and test engineers) are unfamiliar with these forms of inputs and can not acquire the necessary knowledge in a reasonable amount of time in regards to the generated results. SPS assists practitioners by providing mapping between commonly occurring properties and the formal representation of a specific language required for a particular verification tool (based on Dwyer et al. [DAC17]). Dwyer et al. differentiate all SPS into two categories (occurrence and order patterns). This was discussed in short in Subsection 2.3.2 in order to provide an overview of common

specification patterns. Since SPS is used throughout this work, it shall be discussed in more detail here with domain-specific examples.

Occurrence patterns [DAC17]

"[...] the occurrence of a given event/state during system execution"

Universality, Absence, Existence (including Bounded Existence as a variation of Existence pattern)

Order patterns [DAC17]

"[...] relative order in which multiple events occur during system execution"

Response, Precedence (including Response Chain and Precedence Chain as variation of Response, and Precedence)

The five basic patterns are discussed in detail. All variations can be derived from these by combining, extending or nesting the initial five patterns. Descriptions as provided by Dwyer et al. [DAC98, DAC99, DAC17], examples are added for understanding.

Universality ('Always') - P is true

"To describe a portion of a system's execution which contains only states that have a desired property." [DAC17]

Example: CountryCode[US] is true

Absence ('Never') - P is false

"To describe a portion of a system's execution that is free of certain events or states." [DAC17]

Example: SystemVoltage[<0] is false

Existence ('Eventually/Future') - P becomes true

"To describe a portion of a system's execution that contains an instance of certain events or states." [DAC17]

Example: IgnitionSwitch[on] becomes true

Response ('Follows/Leads to') - S responds to P

"To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect." [DAC17]

Example: LowBeamHeadlightLeft[on] responds to LightSwitchPos[on]

Precedence ('Prior to') - S precedes P

"To describe relationships between a pair of states where the occurrence of the first is a necessary pre-condition for an occurrence of the second. We say that an occurrence of the second is enabled by an occurrence of the first" [DAC17]

Example: IgnitionSwitch[on] precedes LowBeamHeadlightLeft[on]

Each of these patterns contains five cases which describes the scope of applicability.

Globally - P is true globally

Given condition is valid for all moments until the system terminates.

Example: CountryCode[US] is true globally

Before R - P is true before R

Given condition is valid for all moments until the condition 'R is true' occurs.

Example: VehicleVelocity[0] is true before HandBrake[Released]

After Q - P is true after Q

Given condition is valid for all moments after the condition 'Q is true' occurs.

Example: Radio[on] is true after IgnitionSwitchPosition[Radio]

Between Q and R - P is true between Q and R

Given condition is only valid for all moments that lie in between the occurrence of 'Q is true' and the occurrence of 'P is true'.

Example: BatteryMode[Charging] is true between ChargingCable[Connected] and BatteryChargingLevel[Full]

After Q until R - P is true after Q until R

Given condition is valid for all moments after 'Q is true' until 'R is true' occurs or if not, until the system terminates.

Example: DirectionIndicatorLeftFront[on] is true after PitManArmPosition[Left] until PitManArmPosition[neutral]

(*Remark: PitManArmPosition refers to the lever or button used in a car to activate the lights for the turn signals.*)

While the scope for 'globally', 'before' and 'after' is obvious, the difference between 'between' and 'after until' is more fine-grained. 'Between' necessarily requires the existence

of both events, a moment where 'Q is true' and a moment where 'R is true', to occur. 'Until' necessarily starts with 'Q is true' but can occur without the existence of an event where 'R is true'. If this event never happens, the given condition remains until the system terminates. The five basic patterns contain the variables P, S and T. The five cases in regards to scope are described through the variables Q and R. There exists a general difference between the two groups of variables. P, S and T are placeholders for system parameters. These variables are attributes and properties of the described system. Q and R are time-based parameters which provide occurrence order for events and limit the scope of a pattern. This differentiation is used in Section 4.3 to generate state machines and separate input parameters for states and transitions. Integration of SPS into the requirements documentation process can be achieved by directly specifying requirements in SPS or by conversion of textual requirements into SPS. These alternatives are discussed in the next two subsections.

3.2.2. Elicitation and Documentation as Text and Conversion to Patterns

This section discusses the process where elicitation and documentation as one task, are separated from the process of conversion of a given textual requirements description into patterns. Subsection 3.2.3 addresses the process of a straight one-step approach. Separation seems beneficial under consideration of two different aspects. Separation of documentation and conversion into two tasks makes both tasks simpler and therefore reduces the risk of false or incorrect semantics in the derived representations. Separation is also the suitable approach for already existing specification documents. In many cases, requirements elicitation is already completed and textual descriptions exist. The urge for formal representations of initial informal textual representations therefore requires only conversion since elicitation and documentation were already performed upfront.

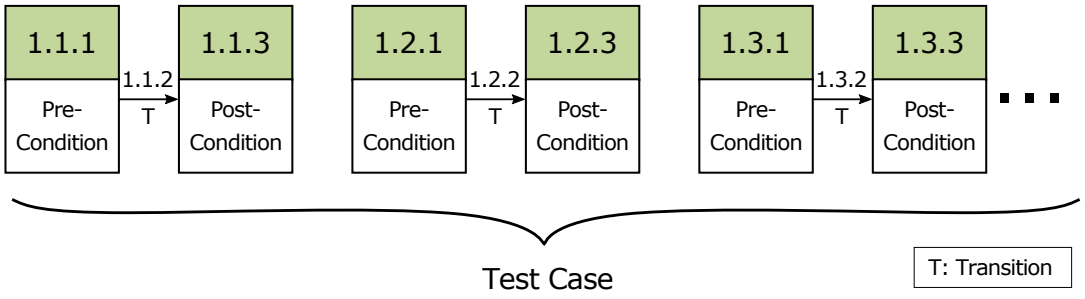


Figure 3.3.: Testing: Test Case Structure and Description, [WHPR17], adapted

Walter et al. [WHPR17] used the separation approach in a case study and found it suitable for conversion of existing test descriptions without any observed occurrences of error introduction. This section will discuss specifically the use case of conversion of preexisting data from a textual form to patterns. The above mentioned case study was performed on test cases. It is therefore suitable to use test cases in this subsection. The data shown in this context stems from an MBC system test specification. By Definition 2.23, a test case consists of mutual test steps. Figure 3.3 shows one test case with three test steps. This structure was characterized in Subsection 2.3.6 as a cycle-free directed tree with 'node valence' ≤ 2. This means one directed branch without forks or cycles. Descriptions discussed in this context do not occur at the test case level but for each test step. Based on Definition 2.25, each test step consists of three elements where each element contains its own description: precondition, trigger and postcondition. Precondition and postcondition are states while trigger is a transition.

Remark. *Nomenclature is: TestCase.TestStep.(State/Trigger)*
(e.g. first state of fifth step in second test case 2.5.1.), Walter et al. [WHPR17].

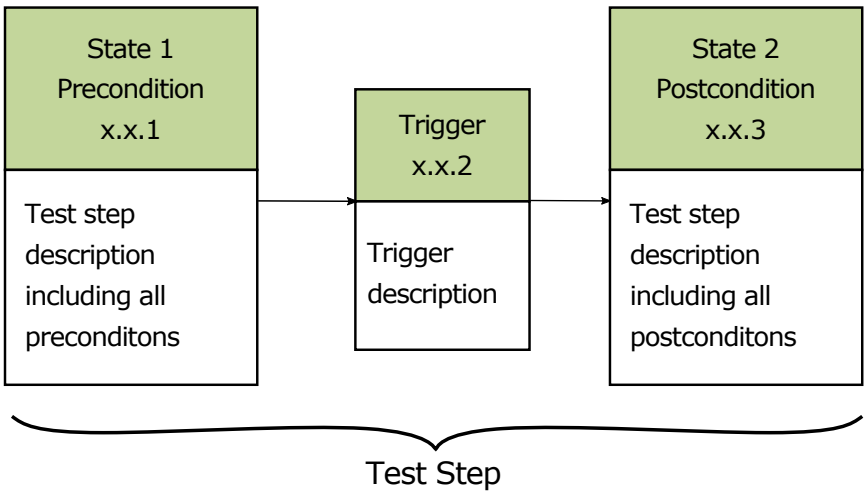


Figure 3.4.: Testing: Test Step Structure and Description, [WHPR17], adapted

The detailed structure of a test step (as used in this example) is shown in Figure 3.4. The precondition trigger and postcondition all contain a textual description. Precondition as defined in Definiton 2.26 describes the state conditions that must be fulfilled by the system before the test step can be executed. Similar, the postcondition, defined in Definition 2.28, describes the state conditions that must be fulfilled after the test step execution in order for the test step to 'pass'. The trigger, defined in Definition 2.27,

describes the cause of the transition between $state_1$ and $state_2$. Table 3.1 shows an example from Walter et al. [WHPR17] for the Outside Light Control (OLC), an MBC system. It contains three test steps from a test case for 'Reverse light functionality' in textual and pattern form.

Table 3.1.: Conversion: Text to Patterns (Example), Walter et al. [WHPR17], adapted)

	Natural Language	Specification Pattern System
1.1.1	Country code is unknown	CountryCode[Unknown] is true at this state*
1.1.2	see **	see **
1.1.3	Country code is US	CountryCode[US] is true globally
1.2.1	No reverse gear before gear is on. Reverse lights are off before reverse gear is on	Gear[Neutral] OR Gear[Forward] AND ReverseLight[Off] are true before Gear[Reverse]
1.2.2	see **	see **
1.2.3	Reverse lights are off	Gear[Neutral] OR Gear[Forward] is true before Gear[Reverse] AND ReverseLight[Off] are true at this state
1.3.1	Ignition switch is off	IgnitionSwitch[Off] AND ReverseLight[Off] are true at this state
1.3.2	see **	see **
1.3.3	Reverse lights stay off	IgnitionSwitch[Lock] AND ReverseLight[Off] are true at this state OR IgnitionSwitch[Radio] AND ReverseLight[Off] are true at this state

Remark. * *SPS was initially designed for requirements. Since tests are described in this context, an additional pattern 'P is true at this state' is introduced in this work. This new pattern does not contain temporal expressions. P has a scope of only one state or transition and therefore only applies at one particular state in this example*

Remark. ** *Steps 1.1.2, 1.2.2 and 1.3.2 represent the transitions. For NL and SPS, the transition information is included in x.x.1 and x.x.3 and therefore x.x.2 is not needed.*

The conversion of textual representations into patterns can be performed in five steps. It is shown with examples of the description for '1.2.1 - $TestCase_1.TestStep_2.State'_1$.

1. Understanding the textual description on the semantic level

   - Before any formalization can occur, the semantic meaning of a textual representation has to be understood. This step is the most critical one in regard to error introduction due to the ambiguous textual representation.

- *'Not reverse gear and reverse lights are off before reverse gear is on'* is already a precise description. It represents a condition where the vehicle shall **not** be in reverse gear while at the same time the reverse lights are off.

2. Extracting system parameters and its particular values from the description.

   - This step converts freely used textual expressions into system parameters. It therefore represents the core formalization step during this process.
   - *'Reverse lights'* represents one parameter '$ReverseLight$'. It seems suitable to selected the parameter type as enum. Its values are On and Off. The more complicated case is *'reverse gear'*. While the parameter 'ReverseGear' could be used, it is also feasible to use 'Gear'. It depends on the system parameter list and previous definitions.

3. Matching (and updating) of the existing system parameter list including values

   - The existing system parameter list has to be checked for already existing, similar parameters. It contains parameter name, type (boolean, enumeration or discrete), physical unit and list of valid values or range.
     Remark. *Representation of system parameters occurs by selecting a parameter and one of its possible values, e.g. parameter = CountryCode; value = US; thus* $CountryCode[US]$.
   - *'*$ReverseLight[Off]$*'* is the first 'parameter/value' pair checked with the system parameter list. The second one is *'Gear'*. The scope of the list is limited to the 'OLC system' domain. System reactions (front, rear and turn signal lights) only differ for three situations: 'forward gear', 'neutral' and 'reverse gear'. Therefore, it seems feasible to define one parameter $Gear$ of type 'enumeration' with entries $Forward$, $Neutral$ and $Reverse$.
   - Parameter: $ReverseLight[On/Off]$, $Gear[Forward/Neutral/Reverse]$

4. Selection of an appropriate specification pattern

   - In this example a suitable SPS pattern is 'universality': 'P is true' with scope 'before R'.
   - *'P is true before R'*

5. Replacing the generic parameters of the pattern with the extracted parameters from the system list

   - After selecting the SPS pattern, it has to be adjusted to the desired expression. Any parameter can be substituted for a combination or alternation of parameters, e.g. $P \longrightarrow P_1\ AND\ P_2$; $P_1 \longrightarrow P_1a \vee P_1b$. In this case, P is substituted into $P_1\ AND\ P_2$. In addition, P_1 is substituted into $P_{1a} \vee P_{1b}$.
   - $P_{1a}\ OR\ P_{1b}\ AND\ P_2$ *is true before* R
   - All abstract parameters P_{1a}, P_{1b}, P_2 and R are replaced with the previously derived parameters.
   - Gear[Neutral] OR Gear[Forward] AND ReverseLight[Off] are true before Gear[Reverse]

This process is performed for every description in isolation. Separation of elicitation & documentation from conversion is suitable for data with high potential of error introduction as well as data already documented textually. The case study showed that separation was particularly useful for test data. Difficulty for test representations lies in tasks which are covered here in $step_2$ and $step_3$ regarding the extraction of measurable quantities and maintaining a list of consistent parameters. The majority of test steps was represented with the 'universality' pattern which allows isolation of each description. This is due to the observed low level of dependencies between steps. $Step_4$ seems more important in the context of requirements.

3.2.3. Elicitation and Documentation as Patterns

In contrast to Subsection 2.3.1, this subsection discusses elicitation with direct documentation in patterns. While both approaches follow similar steps, there exists differences that shall be pointed out. Direct documentation in patterns removes textual documentation. Textual representations might be required for non-technical purposes (e.g. legal, marketing, ...). It is only feasible to use patterns as a single source of documentation when they are accepted as appropriate documentation for such cases. The combination of multiple steps can affect the likelihood of error introduction negatively. It was not observed in any of the case studies in Walter et al. [WHPR17, WSPR18, WMR18, WMS+19], yet the possibility shall be mentioned.

One of the advantages of direct pattern representation without prior expression in natural language, is the reduction of the process by one step. This reduces the required work effort since no conversion has to be performed. In addition, it allows for a better suited and more tailored selection of patterns for a given technical context representation. This is particularly beneficial in the context of requirements where generally more complex relations occur than in test environments. The steps for elicitation and direct documentation in patterns can be broken down in analogy to the five steps of conversion in the previous subsection. $Step_1$ and $Step_2$ differ while $Step_3$, $Step_4$ and $Step_5$ are similar. In this case, the process is shown with an example requirement from the Adaptive Outside Light Control (AOLC), an MBC system. The case study used in this example was initially shown in Walter et al. [WMR18].

1. Extracting the needed system condition or behavior (elicitation)

   - Elicitation is not in the focus of this discussion. For more detailed explanation see Franco [Fra15] and Wong et al. [WMR17].
   - The needed system condition is that for turning the light switch into exterior position, the left and right side low beams should turn on. This is only given in situation where the ignition is turned on.

2. Defining relevant system parameters and values

   - The understanding about the system condition or behavior derived in $Step_1$ must be represented through measurable, controllable quantities. Quantities in this context are parameters with values.
   - Parameter: LowBeamHeadlightLeft[On], LowBeamHeadlightRight[On], LightSwitchPos[Exterior], Ignition[On]

3. Matching (and updating) of the existing system parameter list including values

   - See $Step_3$ of Subsection 3.2.2. Matching new parameters against the existing system parameter list. Boolean parameters usually are simple in processing. Enumerations highly depend on system context and scope.
   - Parameter: LowBeamHeadlightLeft[On/Off], LowBeamHeadlightRight[On/Off], LightSwitchPos[Off, Parking, Exterior], Ignition[On/Off]

4. Selection of an appropriate specification pattern

   - By selecting a particular pattern, the initially free and unrestricted representation is reduced in its expressiveness as mentioned in Subsection 2.3.1. It is suitable for requirements not to split this core task over multiple tasks. In contrast to tests, the variety of patterns and scopes used for requirements is much wider. This step represents the most important step of the direct pattern representation process.
   - In this particular case, it is a design decision that the symmetrical behavior for the left and right sides is combined in one requirement and therefore one pattern in contrast to a separation into two requirements.
   - 'S responds to P after Q'

5. Replacing generic parameters S_1, S_2, P and Q with system list parameters

   - See $Step_5$ above. All abstract parameters are replaced with previously derived parameters, therefore S is substituted with S_1 AND S_2.
   - LowBeamHeadlightLeft[On] AND LowBeamHeadlightRight[On] responds to LightSwitchPos[Exterior] after Ignition[On]

The direct documentation in patterns seems more suitable for complex requirements. $Step_4$ is the key step in this process. The selection and scope of a pattern significantly influences the way requirements are structured, combined and aligned. In its basic form, SPS can certainly be considered a natural language specification, yet combining multiple parameters with AND and OR certainly reduces readability. Overall it can be said that the two shown specification approaches (conversion from text and direct documentation in SPS) both successfully derive work items (requirements and test steps) with structured textual descriptions. This is the foundation for all further steps and serves as the input for the mapping of specification patterns to LTL, which is discussed in the next subsection.

3.3. Specification Patterns to Linear Temporal Logic Expressions

The mapping of (unstructured or structured) text to logic expressions represents a significant step during the formalization process. In Section 2.3, it was discussed that

machine-based inference requires formal representation to reason from the given data. "Change from informal to formal representation leads to an increase in machine readability and reasoning capability while it decreases human readability." The inference capabilities increase drastically once data is represented in mathematical (or logic) form. The question can be raised, why a mapping is required at all. Would it not be feasible to specify requirements in logic expressions directly? While this would work in principle, there are two strong arguments for mapping and against direct specification in logic:

1. Specification in logic is complex. Dwyer et al. [DAC17] suspected that "most programmers are unfamiliar with formal specification languages." Programmers can be seen as representatives for most groups of developers or engineers involved in systems engineering. Therefore, the majority of participants would be excluded from the requirements specification process.

2. Textual representation might be required for non-technical reasons (legal, marketing, ...). In this work, NL and SPS specification shall be considered as textual representations. This was mentioned in the previous section already. An increase in machine readability leads to a decrease in human readability. Mapping solves that problem by providing textual and formal representation, thus serving both needs.

The solution therefore is a text-based specification followed by a formalization in form of a mapping from language space to logic space. Figure 1.2 showed this in the introduction. There is no possibility to derive a mathematical provable mapping between language and logic space. Only heuristic or empirical proven approaches are possible. The following two subsections discuss the case study for Dwyer's SPS patterns with the validation. In addition, abstract patterns and scopes are shown before MBC system examples are introduced. This is combined with a discussion about the differences between test and requirement data in context of 'SPS to LTL' mapping.

3.3.1. Empirical Validation of SPS to LTL Mappings

Dwyer et al. [DAC98, DAC99] performed a case study with an empirical analysis to validate 'SPS to LTL' mappings for the specification patterns derived and developed in their work. These patterns with their mappings are used in this work. The case study contained 555 specifications (requirements) from 35 different sources. "For most we had an expression of the requirement in a specific specification formalism (e.g. LTL).

For many we also had an informal prose description of the requirement." [DAC98] All specifications were manually analyzed and matched to existing SPS patterns in six cases:

A requirement was...

1. matched exactly to a mapping
2. formally equivalent to a mapping (e.g. $\neg \diamond P = \diamond \neg P$)
3. derived from mapping through substitution (e.g. combinations of propositions)
4. a known variant of one of the existing mappings
5. a new variant of an existing mapping (this served as input for extension of SPS)
6. formally described with a clear error; fixing the error led to one of the above cases.

The results in Table 3.2 show matches for all patterns and scopes. The common patterns ('response', 'universality', 'absence') occurred in high numbers (245/119/85). 'Precedence' and 'existence' were found in smaller numbers (26/26). Only the derivatives 'response chain', 'precedence chain' and 'bounded existence' did occur in quantities (8/1/1) that would not allow a generalized claim, that these patterns are validated completely by the study.

Table 3.2.: Case Study: SPS Patterns and Scopes, Dwyer et al. [DAC98]

Pattern	Global	Before	After	Between	Until	Total
Absence	41	5	12	18	9	85
Universality	110	1	5	2	1	119
Existence	12	1	4	8	1	26
Bounded Existence	0	0	0	1	0	1
Response	241	1	3	0	0	245
Precedence	25	0	1	0	0	26
Response Chain	8	0	0	0	0	8
Precedence Chain	1	0	0	0	0	1
Unknown	-	-	-	-	-	44
Total	438	8	25	29	11	555

The validation for the scopes showed that the overwhelming majority used 'global' (438). 'Between' and 'after' occurred in relative moderate numbers (29/25) while 'until' and 'after' only occurred in a few cases (11/8). Four observations towards the results of Dwyer's case study before Dwyer's own conclusion is provided: (1) The majority of patterns was

well reviewed and matched; only the two 'chain' patterns and 'bounded existence' should be treated as not necessarily validated. (2) The scopes are all validated; while 'until' and 'before' only occurred in few cases, these are intrinsically included in the validation of between and after. All scopes except for 'global' are combinations of 'before R' and 'after Q'. (3) The combinations 'pattern × scope' are often not validated. Intuitively this might seem problematic but when investigated closely it becomes clear that pattern and scope can be seen as a combination of two independent logic expressions. Therefore, the isolated validation for each pattern and each scope combined with the argument that addition is unproblematic resolves that problem. (4) The occurrence of unknown is not described in detail. Dwyer et al. [DAC98] mentions errors in specifications. In addition, it is likely that certain requirements (e.g. Non-Functional Requirements) are out of scope for these patterns, which focus on functional requirements.

Table 3.3.: Logic Operators: Nomenclature

	Combination	Alternative	Negation	Implication	Global	Future	Next	Until
Walter	AND	OR	NOT	IMP	G	F	N	U
	$\wedge$	$\vee$	$\neg$	$\longrightarrow$	$\Box$	$\diamond$	$\circ$	U
Dwyer	&	|	!	->	[]	<>	$\circ$	U

Remark. *The original representation of all mappings at [DAC17] contains alternative representations for most logic operators. Mapping between logic operators were introduced in Subsection 2.3.4 and logic operators from Dwyer et al. [DAC17] are shown in Table 3.3.*

Dwyer et al. [DAC98] concluded that from the results of the case study "the answer to [the] question 'Are the specifications generated from the patterns [...] correct?' is yes". In addition "[...] the mappings also underwent testing by running existing finite state verification tools to analyze finite-state transition systems [...]." [DAC17] The case study verified the general mapping of SPS to LTL and the next subsection shows the qualitative mapping with examples from MBC systems.

3.3.2. Qualitative Conversion of Patterns to Logic

This subsection contains 'SPS to LTL' mapping derived by Dwyer et al. and validated through the case study mentioned in Subsection 3.3.1. Mapping for all eleven patterns is shown with the first scope 'globally' in Table 3.4. Mapping 'SPS to LTL' for all five

scopes with pattern 'universality' is shown in Table 3.5. Representation of logic operators in this work differs from the representation used in Dwyer et al. [DAC98, DAC99], which is why a translation table is given in Table 3.3. Mapping 'SPS to LTL' represents the conversion from language to logic space. Figure 1.2 showed that this conversion represents a core step in the formalization process. The different gestalt of the elements in both spaces prevents rule-based mapping. The only feasible approach is an empirical solution with case-based mapping and manual validation. Table 3.4 contains the resulting case-based mappings for one case of each pattern in its structural form and its temporal expression. It can be discussed now which transformation 'NL to SPS' or 'SPS to LTL' is the true conversion from language to logic. In the author's opinion SPS is still considered textual (since it maintains a high readability) and mostly uses free text symbols (exceptions are the logic symbols AND, OR and NOT). In addition, the transformation from 'SPS to LTL' can only be performed based on empirically derived mappings. This indicates a change from one domain (language) to another domain (logic). Therefore, this transformation shall be considered the core transition from language to logic.

Table 3.4.: Mapping: SPS to LTL (all Patterns - Scope: 'Globally'), Dwyer [DAC17]

Pattern	SPS	LTL
Universality	P is true globally	$\Box P$
Absence	P is false globally	$\Box \neg P$
Existence	P becomes true	$\diamond P$
Bounded Existence	P becomes true while transitions to P-state occur at most 2 times	$(\neg PW(PW(\neg PW(PW\Box\neg P)))$ where $PWQ = \Box P \vee (PUQ)$
Response	S responds to P globally	$\Box(P \longrightarrow \diamond S)$
Response Chain (2 stimulus, 1 resp.)	P responds to S, T	$\Box(S \wedge \circ \diamond T \longrightarrow \circ(\diamond(T \wedge \diamond P)))$
Response Chain (1 stimulus, 2 resp.)	S, T responds to P globally	$\Box(P \longrightarrow \diamond(S \wedge \circ \diamond T))$
Precedence	S precedes P globally	$\diamond P \longrightarrow (\neg PU(S \wedge \neg P))$
Precedence (2 causes, 1 effect)	S, T precedes P globally	$\diamond P \longrightarrow (\neg PU(S \wedge \neg P \wedge \circ(\neg PUT)))$
Precedence (1 cause, 2 effects)	P precedes (S, T) globally	$(\diamond(S \wedge \circ \diamond T)) \longrightarrow ((\neg S)UP)$
Constrained Chain Patterns	S, T without Z responds to P globally	$\Box(P \longrightarrow \diamond(S \wedge \neg Z \wedge \circ(\neg ZUT)))$

The full list of all 55 mappings (11 patterns × 5 scopes) can be found in Appendix A as well as the original source website by Dwyer et al. [DAC17]

When investigating LTL expressions in Table 3.4, a couple of observations can be made. 'Universality', 'absence' and 'existence' each contain one atomic temporal expression ($\Box$ global / $\diamond$ future) (in case of 'absence' an additional negation) combined with a single parameter. Patterns are used to express single conditions against the system for a certain time frame defined through the scope. 'Response' and 'precedence' put two parameters in a time-relation. Both are still relatively simple and only contain two temporal operators ($\Box$ global and $\diamond$ future, and $\diamond$ future and U until). 'Response' and 'precedence' are the basic patterns that serve as building blocks for the more complex patterns 'response chain', 'precedence chain' and 'constrained chain patterns'. All of these put three or four parameters in relation and repeat complex temporal logic constraints. In contrast to structural relation between 'response' and 'precedence' with its more complex chain derivatives, the case for 'existence' is different. 'Existence' and 'bounded existence' share only commonalities in its name, semantic meaning and pattern expression. Its temporal expressions vary tremendously. In temporal logic, 'bounded existence' should be treated as an independent pattern, not as a derivative.

Table 3.5.:
Mapping: SPS to LTL (all Scopes - Pattern: 'Universality'), Dwyer et al. [DAC17]

	SPS	LTL
Globally	P is true globally	$\Box P$
Before R	P is true before R	$\diamond R \longrightarrow (P\ U\ R)$
After Q	P is true after Q	$\Box\ (Q \longrightarrow \Box\ (P))$
Between Q and R	P is true between Q and R	$\Box\ ((Q \wedge \diamond R) \longrightarrow P\ U\ R)$
After Q until R	P is true after Q until R	$\Box\ (Q \longrightarrow P\ U\ (R \vee \Box\ (P)))$

Scope related logic occurs before and after the logic expression, generated based on each particular pattern. The pattern is constant and the scope can be adjusted and exchanged accordingly. This was used as an argument in the previous subsection to justify why the validation of patterns and scopes in Table 3.2 does not require all combinations (pattern *times* scope) that can be performed in isolation. (Validation of all patterns, validation of all scopes and argumentation that all combination arise from addition of a valid pattern and a valid scope.) 'Before R' and 'After Q' are the basic scopes while 'between Q and R' and 'after Q until R' represent derivatives of the combination of these two basic patterns. It was discussed beforehand that representations in SPS and LTL differ for tests and requirements. Therefore, both shall be addressed separately. The example of tests from Subsection 3.2.2 is continued in Table 3.6.

Table 3.6.:
Conversion: SPS to LTL - Test Case (Example), Walter et al. [WHPR17], adapted

	Specification Pattern System	Linear Temporal Logic
1.1.1	CountryCode[Unknown] is true at this state *	CountryCode[Unknown]
1.1.2	see **	CountryCode[US]
1.1.3	CountryCode[US] is true globally	$\square$ CountryCode[US]
1.2.1	(Gear[Neutral] OR Gear[Forward]) $\wedge$ ReverseLight[Off] are true before Gear[Reverse]	$\diamond$ Gear[Reverse] $\longrightarrow$ (((Gear[Neutral] $\vee$ Gear[Forward]) $\wedge$ ReverseLight[Off]) U Gear[Reverse])
1.2.2	see **	Gear[Reverse] U $\neg$ Gear[Reverse]
1.2.3	Gear[Neutral] OR Gear[Forward] is true before Gear[Reverse] AND ReverseLight[Off] are true at this state	$\diamond$ (Gear[Reverse] $\wedge$ ReverseLight[Off]) $\longrightarrow$ ((Gear[Neutral] $\vee$ Gear[Forward]) U (Gear[Reverse] $\wedge$ ReverseLight[Off]))
1.3.1	IgnitionSwitch[Off] AND ReverseLight[Off] are true at this state*	IgnitionSwitch[Off] $\wedge$ ReverseLight[Off]
1.3.2	see **	IgnitionSwitch[Lock] $\vee$ IgnitionSwitch[Radio]
1.3.3	IgnitionSwitch[Lock] AND ReverseLight[Off] are true at this state OR IgnitionSwitch[Radio] AND ReverseLight[Off] are true at this state	(IgnitionSwitch[Lock] $\wedge$ ReverseLight[Off]) $\vee$ (IgnitionSwitch[Radio] $\wedge$ ReverseLight[Off])

Remark. * *SPS was initially designed for requirements. Since tests are described in this context, an additional pattern 'P is true at this state' is introduced in this work. This new pattern does not contain temporal expressions. P has a scope of only one state or transition and therefore only applies to one particular state in this example*

Remark. ** *Steps 1.1.2, 1.2.2 and 1.3.2 represent the transitions. For NL and SPS, the transition information is included in x.x.1 and x.x.3 and therefore x.x.2 is not needed.*

The requirement example from Subsection 3.2.3 is further developed below the test examples. Test cases commonly consists of isolated simple statements without many complex dependencies. The isolation of statements becomes particularly obvious when investigating the transitions 1.1.2, 1.2.2 and 1.3.2. Methodically, a transition represents a condition that must be fulfilled to change the current system state. Generally such a condition consists of one or multiple time-independent, isolated parameters. If a transition condition contains time-dependent relations between parameters, this transition is not atomic. Such transitions should be adjusted from one *transition* to a *transition-state-transition*

structure. While time-related dependencies between multiple parameters in a transition are methodically not correct, the parameter that is adjusted to transition the system remains in this particular value until defined otherwise. Therefore, the LTL expression for a single parameter in a transition is commonly: 'parameter[value] U $\neg$ parameter[value]'.

Overall, test data tends to contain less complicated relations. In Table 3.6, three examples of LTL occur: (1) State 1.1.3 contains a global condition for CountryCode[US]. This is a common use case for test data. The first test step provides an initializing setup for the consecutive test steps. To achieve this, postcondition x.1.3 contains conditions that apply globally as boundary conditions for all test steps of the test case. CountryCode[US] is an example for such initializing. Conditions that are stable over all consecutive test steps contain a '$\square$ global' operator. Conditions that change eventually are usually expressed through 'U until' operator combined with an ending condition. (2) Second occurrence of temporal logic is state 1.2.1. It describes a condition that, besides the current state, is affecting the consecutive transition and the following postcondition. In case Gear[Reverse] is occurring, certain conditions must be fulfilled beforehand. (3) The third occurrence of temporal logic is state 1.2.3. The 'global universality' pattern is used and extended for $P = P_1 \vee P_2$ as well as $R = R_1 \wedge R_2$. The condition is also related to the previous transition. In case the transition Gear[Reverse] is performed and ReverseLight[Off] is true, implications for Gear[Neutral] $\vee$ Gear[Forward] follow.

The case study of Walter et al. [WHPR17] used four patterns to express all test cases:

(a) Global Setup Condition - Case (1) is a typical temporal occurrence of a setup condition that persists during all test steps, or when expressed with 'U until' operator instead of '$\square$ global' operator, persists until actively changed.

(b) Transition - A transition occurs with α U $\neg$ α, a system parameter is changed and remains until specified otherwise.

(c) Test Scenario (local) - Precondition and postcondition are expressed in a logic statement without temporal expressions and thus only affecting the local state

(d) Test Scenario (global) - A complex test scenario is expressed at one state (precondition or postcondition of a test step). In this case, the relations and system reactions are expressed without direct consideration of the test case structure. Case (2) and (3) are examples for more complex conditions that affect multiple states.

The common forms of use of patterns differs for test cases and requirements. Form (b) does not occur in requirements. In general, requirements express global conditions or define rather complex relations between parameters. As described in Subsection 2.2.1, requirements therefore can be separated into two common forms. This separation manifests itself in the choice of specification patterns that seem suitable for the description of these types of requirements.

(a) Non-functional requirements in form of constraints that provide a static condition without complex time relations for a certain scope. These are similar to case (1) for tests. These are usually described with 'universality', 'absence' and 'existence'.

(b) Functional requirements which describe a variety of time-related constraints and conditions. These are expressed through 'response', 'precedence', all its derivatives as well as 'bounded existence'.

The example for a functional requirement (case (b)) shown in Subsection 3.2.3 with its mapping into LTL is represented in the following way:

Table 3.7.:
Conversion: SPS to LTL - Requirement (Example), Walter et al. [WHPR17], adapted

	Pattern	LTL
Abstract	S responds to P after Q	$\square(Q \longrightarrow \square\ (P \longrightarrow \diamond\ S))$
Concrete	LowBeamHeadlightLeft[On] $\wedge$ LowBeamHeadlightRight[On] responds to LightSwitchPos[Exterior] after Ignition[On]	$\square(Ignition[On] \longrightarrow \square\ (LightSwitchPos[Exterior] \longrightarrow \diamond\ (LowBeamHeadlightLeft[On] \wedge LowBeamHeadlightRight[On])))$

The 'response' pattern with scope 'after Q' puts three or four parameters (since parameter S is separated into $S_1 \wedge S_2$) in relation to each other. The temporal logic that describes the relations represents a compact representation of the requirement. For analysis and consecutive work steps, a less compact, more explicit representation is often beneficial. Section 3.4 and Section 3.5 discuss the conversion from LTL to local FOL for a more explicit representation.

3.4. First Order Logic - Special Modeling Structures

This section discusses the mapping of temporal logic expressions towards FOL under consideration of the underlying modeling data structure. A special focus is on directed

cycle-free tree structures while Section 3.5 generalizes this approach to directed cyclic graphs. The reasoning behind the reduction of temporal logic to FOL is explained first. In Subsection 3.4.1, the core idea of this processing step is shown. This centers around consideration of the underlying modeling data structure when converting temporal logic into FOL. For directed cycle-free tree structures, a rule-based mapping is shown for all occurring temporal logic operators. Examples from the case study performed on the formalization of test data by Walter et al. [WHPR17] are shown and discussed in Subsection 3.4.2.

The representation of data in form of LTL is useful as a means of formalization. LTL expressions can be derived from SPS patterns through case-based mappings as shown in Section 3.3. LTL descriptions are generally compact. One statement can affect many or all existing states in a system, e.g. $\Box\ CountryCode[US]$. While the scope of a 'global' statement is easy to process, statements including 'next' require knowledge about the sequential order of states. It is necessary to know which state from a given 'current' state is in fact the 'next' state. The situation is similar for the operators 'until' and 'future': Particular knowledge about the path in a 'state-transition-state-...' structure is required. Description of the 'current' state depends on its surrounding states. For example if B is not true when the current state is reached, in a situation where '$A\ U\ B$' is given, A is true at the current state. In situations where B is fulfilled beforehand, the condition '$A\ U\ B$' is not relevant for the current state.

The complex, recursive construction of local state descriptions does not cooperate well with the need to perform data analysis. To derive meaningful knowledge about the system, each state description is required. A fully local representation seems suitable. FSM by its definition provides a time-independent local representation. For temporal logic representations, *stateDescription* is a function of 'local description' and all occurring 'global descriptions'. Global descriptions in this case are structure-dependent. Structure dependency therefore is one of the key factors to consider for decoupled, independent local descriptions. Subsection 2.3.6 described the structures that exist in this context. The special form of directed cycle-free one-branch trees is used in the next subsection to derive a rule-based set of mappings from LTL to decoupled local FOL descriptions. Such descriptions are not compact but locally complete and independent of all other local descriptions. This improves analysis capabilities tremendously.

3.4.1. Data Structure as an Enabler to Map LTL Expressions to FOL

It became obvious why local representations in FOL are more suitable for analysis purposes. To achieve local representations, temporal logic has to be removed and replaced by a FOL. It was described that temporal logic representations are tightly connected to data structure of the underling model. The key to decoupled, local representations is therefore the consideration and inclusion of the underlying model data structure. This subsection explains the approach for the special case of forward-directed cycle-free one-branch trees with a rule-based mapping for each temporal logic operator. Subsection 3.5.1 generalizes this for directed cyclic graphs with a case-based mapping 'LTL expression to FOL'. The structure of a directed tree with only one branch is shown in Figure 3.5.

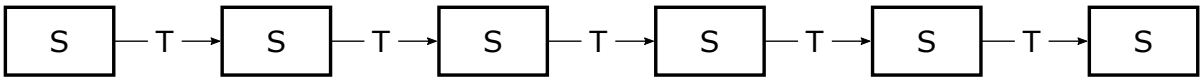


Figure 3.5.: Graph Representation: One Branch Directed Tree - General Structure

Each element in Figure 3.5 represents a node or a state, while the connections represent edges or transitions. The simple structure shall be adapted slightly to the data structure actually occurring in test data. Figure 3.3 showed the methodical structure of one test case, consisting of n test steps. A test step contains two states (precondition (x.x1) and postcondition (x.x.3). Precondition and postcondition are connected through a transition (x.x.2). In real data sets, postcondition of $test\_step_i$ (x.i.3) and precondition of $test\_step_{i+1}$ (x.(i+1).1) are often identical. This is methodically useful but not necessarily given for all data sets. Therefore, a non-methodical link is added in the structure as shown in Figure 3.6. In fact, this converts the structure in fact back into the structure shown in Figure 3.5, which is a simple 'state-transition-. . . '-repetition.

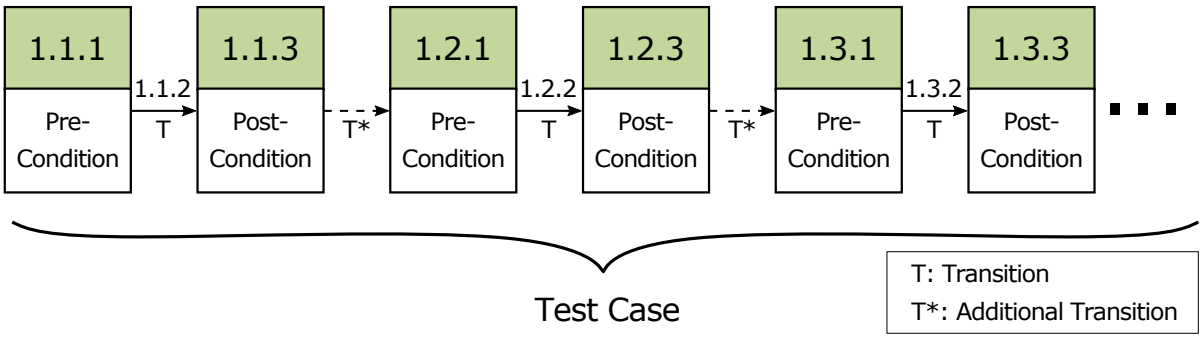


Figure 3.6.: Testing: Test Case Structure - Link-Extension, [WHPR17], adapted]

Remark. *The graphs discussed in Chapter 4 are adapted further. An edge in a UML-graph cannot contain attributes the same way nodes can, which is why transitions are modeled as 'transition-node-transition' structures where the node contains the transition description and further attributes.*

Based on the given structure of a directed cycle-free one-branch tree, a rule-based mapping for each of the selected temporal logic operators is shown. The general definition for $\Box$ 'global', $\circ$ 'next', $\diamond$ 'future' and U 'until' was provided in Subsection 2.3.4. The principles of the mapping from LTL to FOL are described in various sources. The description of Artale [Art10] is the one considered in particular for this work. Each temporal logic operator is converted into FOL through a time-discrete state-wise representation. All states are ordered through the directed tree structure. This structural order includes a temporal order which is the basis for the conversion of LTL to FOL.

Remark. *Generally all LTL can be expressed with S^* (strict 'since') and U^* (strict 'until') defined by Kamp [Kam68]. If only forward-directed events are considered, this reduces to U^* only. For minimal mathematical representation of LTL, U^* is sufficient and complete, yet for practical reasons the three other operators are shown as well. All further operators can be used as long as conversion to a representation with U^* exists. For simplification, U^* will be expressed as U from now on.*

Conversion of $\circ$ 'next'

'Next' affects the next consecutive state from the current state forward. Figure 3.7 shows the LTL representation with its equivalent time-discrete representation in FOL.

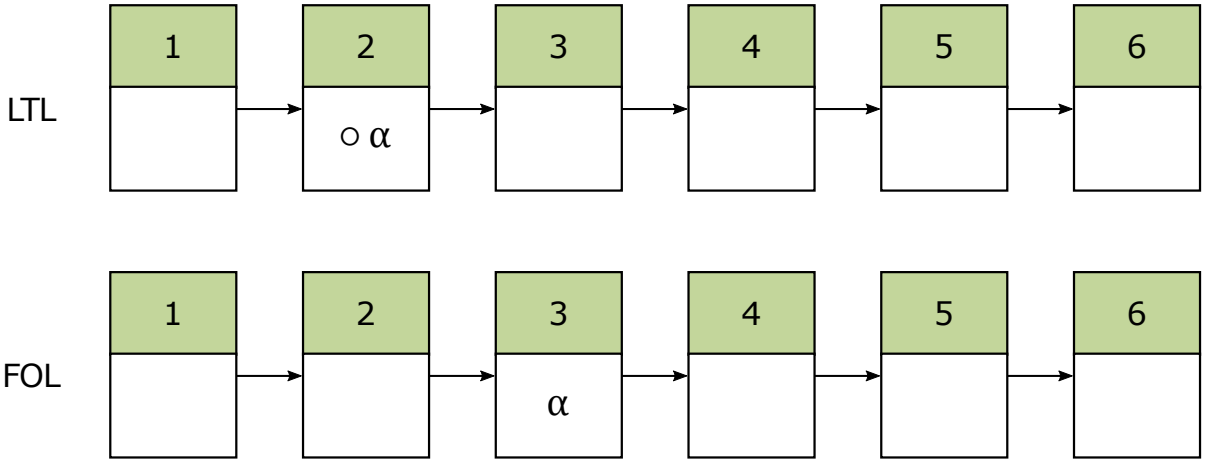


Figure 3.7.:
Logic Representation: Operator 'Next' $\circ$ (Time-Discrete), Artale [Art10], adapted

$State_2$ contains the expression of '$\circ\ \alpha$'. The methodical meaning is that from $state_2$, the next consecutive state contains expression α. Considering conversion of 'next', the data structure must be static. In dynamic and non-time-discrete structures, the next state is not clearly defined, which is why use of 'next' is often avoided. For the given structure and the premise that the structure cannot be changed by adding additional states in between or rearranging the order, 'next' is unproblematic. One formal description of a logic expression is to show each state with its occurring partial expression in FOL. A terminology to order and express all states in one formula is shown. X is introduced to indicate the next step from the given state. Recursively applying this principle with '$X\ (X\ \alpha)$', '$X\ (X\ (X\ \alpha))$', ... enables an unambiguous structuring of all states. This is equivalent to the expression $\circ$, but a structural representation with X is preferred to explain the rule-based mapping.

$$\circ\ \alpha = X\alpha \tag{7}$$

Representation in equation 7 allows a state-wise expression in FOL. This becomes more obvious when considering equation 8. $X\ alpha$ represents the condition that applies at the state that is 'next' state from the 'current' state while $X\ (X\ alpha)$ would represent a condition for 'next' state from the perspective of 'next' state from the 'current' state.

Conversion of $\square$ 'global'

'Global' affects the current state as well as all consecutive (or future) states. Figure 3.8 shows the LTL representation with its equivalent time-discrete representation in FOL.

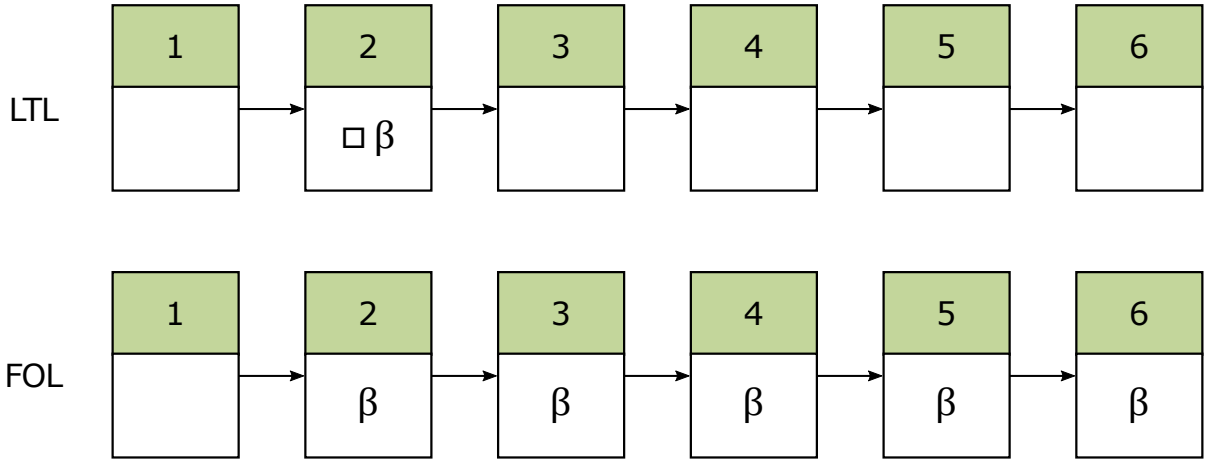


Figure 3.8.:
Logic Representation: Operator 'Global' $\square$ (Time-Discrete), Artale [Art10], adapted

$State_2$ contains the expression of '$\Box\ \beta$'. The methodical meaning is that from $state_2$ on, all consecutive states contain the expression β. The explicit local representation in FOL makes it possible for the isolated consideration of a single state representation to contain the full expression. No other states or structural considerations are required.

$$\Box\beta = \beta \wedge (X(\beta \wedge X(\beta \wedge X(\beta \wedge \ldots)))) \tag{8}$$

For n states, $n-1$ recursive occurrences of '$X(\beta \wedge \ldots)$' are needed.

$$\beta \wedge (X(\beta \wedge X(\beta \wedge X(\beta \wedge \ldots)))) = \beta \wedge X\beta \wedge XX\beta \wedge XXX\beta \wedge \ldots \tag{9}$$

In Equation 9 the recursive representation is converted into a less compact form with improved readability. The state-wise structure becomes visible.

Conversion of U 'until'

'Until' affects the current state and all consecutive states until the stop condition occurs. Figure 3.9 shows the LTL representation with its equivalent time-discrete representation in FOL.

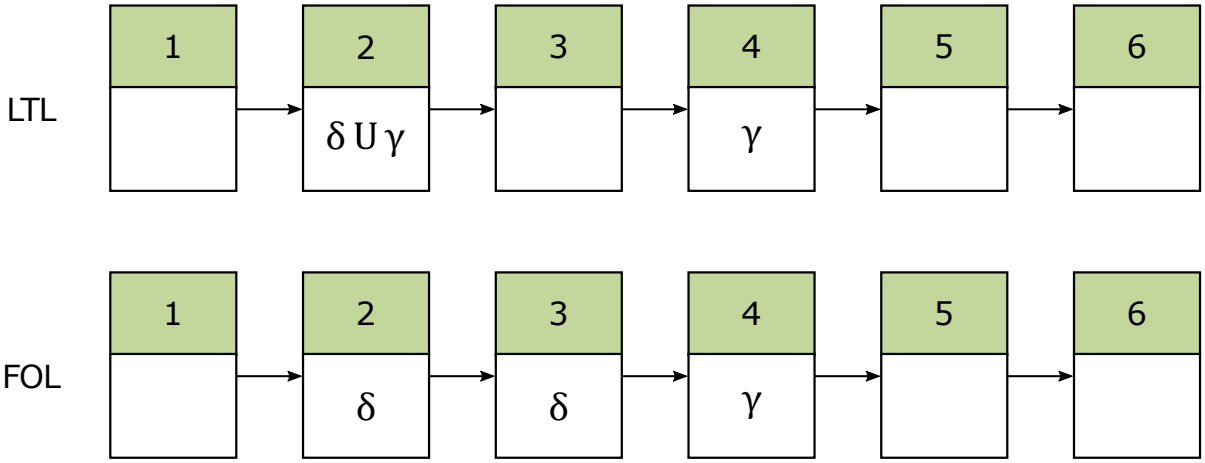


Figure 3.9.:
Logic Representation: Operator 'Until' U (Time-Discrete), Artale [Art10], adapted

$State_2$ contains the expression of '$\delta\ U\ \gamma$'. The methodical meaning is that $state_2$ and all consecutive states contain the expression v until the expression γ occurs. γ appears in $state_4$ thus $state_2$ and $state_3$ contain δ while $state_4$ contains γ.

$$\delta U \gamma = \delta \wedge X\delta \wedge X(X\gamma) \tag{10}$$

The formula shown in Equation 10 represents the particular example shown in Figure 3.9. The current state contains δ, the next step (X) contains δ as well, while the next consecutive step (indicated through XX) fulfills the stop condition *gamma*. For cases where γ never occurs, δ will hold for all times (until the last state). Another special case for 'until' is represented with $\delta\ U\ \neg\delta$. Methodically, such a condition represents a 'global' condition which can be actively adjusted and therefore is only valid until actively changed.

Conversion of $\diamond$ 'future'

'Future' affects one of the consecutive state from the current state to the end state. Figure 3.9 shows the LTL representation with its equivalent time-discrete representation in FOL. The complication for the 'future' operator is that by its definition it is unclear, when exactly the condition is fulfilled. The operator only describes the fact that for n states, the condition will be true at the latest at $state_n$ (last state). Further conditions are required to allow a specific mapping of the operator to the particular state where it is true at first. The definition in this context is, unless otherwise specified, the last possible state (end state) will be the state were the condition occurs.

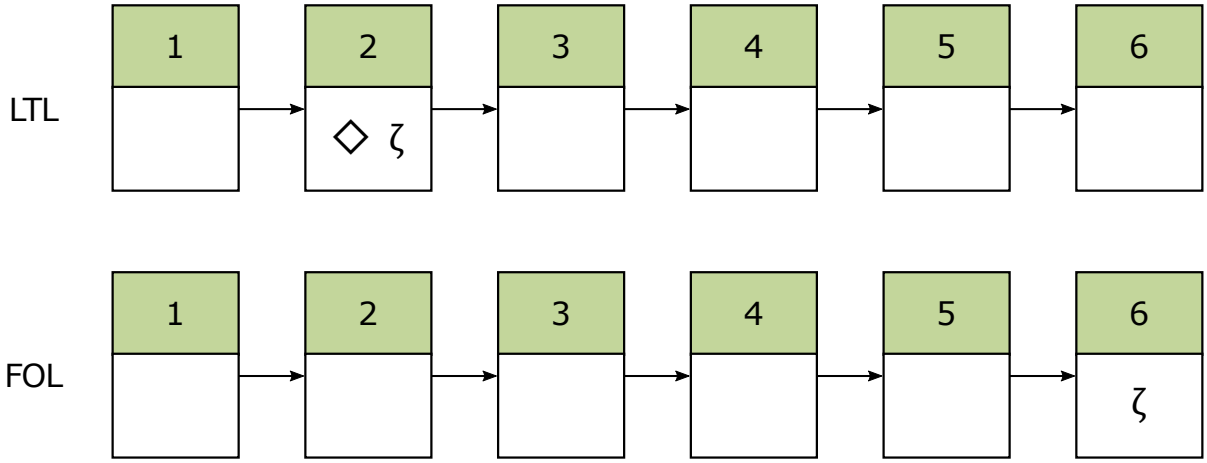


Figure 3.10.:
Logic Representation: Operator 'Future' $\diamond$ (Time-Discrete), Artale [Art10], adapted

$State_2$ contains the expression of $\diamond\ \zeta$. The methodical meaning is that from the current state (here $state_2$) on, at least one state fulfills the condition. Since no further states are defined and no additional conditions exist, the last state ($state_n$) (here $state_6$) contains

ζ. For simplification of representation, Figure 3.10 shows ζ at state 6. Yet only further conditions can determine whether the condition is already fulfilled at state 3, 4 or 5. In this case, ζ is true at that particular state but not necessarily at state 6.

$$\diamond \zeta = X(X(X(X\zeta))) \tag{11}$$

The formula shown in Equation 11 represents the example from Figure 3.10. ζ occurs at the last state $X(X(X(X)))$. There exists no additional condition that would influence the occurrence of ζ in a previous state. As mentioned, 'future' is specifically defined in this context. When no additional information is given, 'future' affects the last state in a particular structure, shown here as $X(X(X(X\zeta)))$. The given reduction rules for the isolated LTL expressions can be extended through conjunctures $\wedge$ AND, $\vee$ OR, $\neg$ NOT and $\longrightarrow$ IMP. In particular $\wedge$ AND, $\vee$ OR and $\longrightarrow$ IMP allow the combination of multiple LTL expressions into more complex expressions. While the conversion of such complex LTL expressions requires more steps, it can always be resolved by recursively applying the four conversion rules for $\Box$ 'global', $\circ$ 'next', $\diamond$ 'future' and U 'until'. This subsection showed locally complete FOL expressions and examples for the reduction of LTL to FOL are shown in Subsection 3.4.2 with a case study initially published in Walter et al. [WHPR17].

3.4.2. Mapping LTL on Directed One-Branch Trees

Subsection 3.4.1 provides abstract rule-based mappings for all four temporal logic operators previously defined for this work. $\Box$ 'global', $\circ$ 'next' and U 'until' can fully describe a condition, while a stand alone expression with $\diamond$ 'future' is generally ambiguous. Abstract mappings are now applied against the data from the case study (Walter et al. [WHPR17]). This process contains four steps. At first, (if possible) each temporal expression is locally simplified. This means 'next' and 'global' expressions are converted into FOL. The underlying model data structure in form of all consecutive steps is considered. Separation between different states occurs through X (as introduced in Subsection 3.4.1). The second step removes all X expressions and moves each expression to its correct local state. This creates independent local representations in FOL. Conditions containing 'next' and 'global' are independent of other expressions and thus can be processed and moved without further consideration.

Table 3.8.:
Conversion: LTL to FOL - Test Case, Step 1 + 2 (Example), Walter et al. [WHPR17])

	LTL	FOL (Concatenation)	FOL (local)
...	...	...	...
1.1.3	$\square$ CountryCode[US]	CountryCode[US] $\wedge$ X CountryCode[US] $\wedge$ XX CountryCode[US] $\wedge$ XXX CountryCode[US] $\wedge$ $XXXX$ CountryCode[US] $\wedge$ $XXXXX$ CountryCode[US] $\wedge$ $XXXXXX$ CountryCode[US]	CountryCode[US]
1.2.1	...	...	CountryCode[US]
1.2.2	...	...	CountryCode[US]
1.2.3	...	...	CountryCode[US]
1.3.1	...	...	CountryCode[US]
1.3.2	...	...	CountryCode[US]
1.3.3	...	...	CountryCode[US]

In contrast to 'next' and 'global', 'until' and 'future' are affected by existing conditions or state expressions. Conditions containing 'until' and 'future' are processed in step 3, after comparison to the derived local statements from step 2. The processing is shown with the examples from Subsection 3.3.2. Step 1 and step 2 (shown in Table 3.8) convert 'next' and 'global' expressions into a concatenation of FOL expressions and further into local FOL statements. State 1.1.3. contains the 'global' condition $\square$ CountryCode[US] which can be separated into a state-wise concatenation of FOL expressions shown in Column 'FOL concatenation'. The example contains nine states and transitions. For the remaining states and transitions from 1.1.3 to 1.3.3, seven statements are required, thus CountryCode[US] occurs seven times in the state-wise representation. Column 'FOL local' processes the state-wise expressions from column 'FOL concatenation' to its respective state. Each X moves a statement one state or transition further. The meaning of $\square$ CountryCode[US] is to be a valid condition for every consecutive step. This meaning is maintained throughout the representation given with the state-wise FOL expressions. When step 1 and step 2 are completed, 'until' and 'future' expressions can be processed. Step 3 as well as step 4 are context related.

The scope of any 'until' or 'future' statement depends on the surrounding conditions. 'Until' statements must be processed before 'future' statements. The reason therefore lies

in the meaning of the two operators 'until' and 'future'. While 'until' describes conditions, statements including 'future' methodically serve as an 'if this then that' statement. Thus 'future' requires all other statements to exist prior to its check up for the 'if' part.

Table 3.9.:
Conversion: LTL to FOL - Test Case, Step 3 + 4 (Example), Walter et al. [WHPR17])

	LTL	FOL (U local)	FOL ($\diamond$ local)
...	...	...	...
1.1.3	...	...	...
1.2.1	$\diamond$ Gear[Reverse] $\longrightarrow$ (((Gear[Neutral] $\vee$ Gear[Forward]) $\wedge$ ReverseLight[Off]) U Gear[Reverse])	...	(Gear[Neutral] $\vee$ Gear[Forward]) $\wedge$ ReverseLight[Off]
1.2.2	Gear[Reverse] U $\neg$ Gear[Reverse]	Gear[Reverse]	Gear[Reverse]
1.2.3	...	Gear[Reverse]	Gear[Reverse]
1.3.1	...	Gear[Reverse]	Gear[Reverse]
1.3.2	...	Gear[Reverse]	Gear[Reverse]
1.3.3	...	Gear[Reverse]	Gear[Reverse]

Examples are shown in Table 3.9. The LTL expression for transition 1.2.2 can be processed in step 3 similar to the 'global' CountryCode[US] statement in Table 3.8. Since no '$\neg$ Gear[Reverse]' occurs, Gear[Reverse] is valid for all consecutive states. Step 4 considers the 'future' statement. Based on the occurrence of Gear[Reverse], the 'future' condition in state 1.2.1 is fulfilled and can therefore be converted to a local statement at the correct state. '(Gear[Neutral] $\vee$ Gear[Forward]) $\wedge$ ReverseLight[Off]' occurs from the given state 1.2.1 until Gear[Reverse] is given. If there would exist additional states between 1.2.1 and 1.2.2, '(Gear[Neutral] $\vee$ Gear[Forward]) $\wedge$ ReverseLight[Off]' would be maintained throughout all in-between states. Conversion of all states and transitions considering all four steps is shown in Table 3.10 when showing the full process example. In this example, the FOL representation is in fact a propositional logic since no qualifies are used. The shown example continues the example from Subsection 3.3.2. Processing for each particular temporal logic operator was shown in Table 3.8 and Table 3.9. State 1.1.2 contains an adjustable condition CountryCode[US]. Since state 1.1.3 overrules the 'until' condition with a 'global' condition, each consecutive state contains CountryCode[US]. State 1.2.1 contained a 'future' condition which is fulfilled in transition 1.2.2. Therefore, the implication for state 1.2.1 follows as: 'Gear[Neutral] $\vee$ Gear[Forward]'. Transition 1.2.2 contains an 'until' condition and since Gear[Reverse] is never changed afterwards, it affects all consecutive states. The 'future' condition 'Gear[Reverse] $\wedge$ ReverseLight[Off]' in state 1.2.3

is fulfilled in state 1.3.1 but the resulting condition 'Gear[Neutral] $\vee$ Gear[Forward]' is contradicting and overruled by the previously applied 'until' condition Gear[Reverse]. As a result, 'Gear[Neutral] $\vee$ Gear[Forward]' has no consequence. Transition 1.3.2. contains an alternation of 'IgnitionSwitch[Lock] $\vee$ IgnitionSwitch[Radio]', each with a singular 'until' contingency. Since neither parameter is changed afterwards, the 'until' condition only affects 1.3.3 where the two alternative conditions are maintained.

Table 3.10.:
Conversion: LTL to FOL - Test Case, All Steps (Example), Walter et al. [WHPR17])

	Linear Temporal Logic	First Order Logic
1.1.1	CountryCode[Unknown]	CountryCode[Unknown]
1.1.2	CountryCode[US] U $\neg$ CountryCode[US]	CountryCode[US]
1.1.3	$\Box$ CountryCode[US]	CountryCode[US]
1.2.1	$\diamond$ Gear[Reverse] $\longrightarrow$ (((Gear[Neutral] $\vee$ Gear[Forward]) $\wedge$ ReverseLight[Off]) U Gear[Reverse])	(Gear[Neutral] $\vee$ Gear[Forward]) $\wedge$ ReverseLight[Off] $\wedge$ CountryCode[US]
1.2.2	Gear[Reverse] U $\neg$ Gear[Reverse]	CountryCode[US] $\wedge$ Gear[Reverse]
1.2.3	$\diamond$ (Gear[Reverse] $\wedge$ ReverseLight[Off]) $\longrightarrow$ ((Gear[Neutral] $\vee$ Gear[Forward]) U (Gear[Reverse] $\wedge$ ReverseLight[Off]))	CountryCode[US] $\wedge$ Gear[Reverse]
1.3.1	IgnitionSwitch[Off] $\wedge$ ReverseLight[Off]	IgnitionSwitch[Off] $\wedge$ ReverseLight[Off] $\wedge$ CountryCode[US] $\wedge$ Gear[Reverse]
1.3.2	(IgnitionSwitch[Lock] U $\neg$ IgnitionSwitch[Lock]) $\vee$ IgnitionSwitch[Radio]) U $\neg$ IgnitionSwitch[Radio])	(IgnitionSwitch[Lock] $\vee$ IgnitionSwitch[Radio]) $\wedge$ CountryCode[US] $\wedge$ Gear[Reverse]
1.3.3	(IgnitionSwitch[Lock] $\wedge$ ReverseLight[Off]) $\vee$ (IgnitionSwitch[Radio] $\wedge$ ReverseLight[Off])	(IgnitionSwitch[Lock] $\vee$ IgnitionSwitch[Radio]) $\wedge$ ReverseLight[Off] $\wedge$ CountryCode[US] $\wedge$ Gear[Reverse]

Overall, conversion of LTL expressions to FOL achieved a local representation of all conditions for states and transitions. Each representation is locally complete. There exist no structure-related contingencies. All recursive dependencies that existed in LTL form, are resolved in FOL. It is obvious that generally such representations are not compact, yet local representation allows for a much better postprocessing and data analysis. The data represents each state and transition with an (almost) unambiguous statement.The last step towards an unambiguous representation is to convert the FOL to a normal form.

This is described in Section 3.6. Prior to the conversion to a normal form, the generalization of 'LTL to FOL' conversion is given in Section 3.5. This discusses the case-based mapping for LTL expressions in directed cyclic graph structures to FOL expressions.

3.5. First Order Logic Expressions - Generalized Modeling Structures

The usefulness of FOL representations was laid out in the previous section. While test cases can be represented in directed cycle-free structures, requirements need more complex modeling structures. From a graph and data structure standpoint, the differences were discussed in Subsection 2.3.6. Subsection 3.5.1 will address the methodical differences between test and requirements data. A formal representation for requirements are finite state machines (FSM). Detailed implications on the data structure and possible mapping approaches for FSM are addressed. The goal of this section is to provide a mapping from temporal logic to FOL for directed cyclic graphs. While the mapping for directed cycle-free one-branch trees was rule-based, the generalized approach will be case-based for all occurring LTL expressions derived from SPS.

3.5.1. Extending Modeling Structures from Trees to Graphs

The solution space for a system is initially represented through all possible states the system can be in. States arise from the combination of all parameters with all its possible values. Each state represents a node in the system graph as described in Subsection 2.3.4. Nodes are connected through edges which methodically represent transitions that allow the system to change the 'current state' of the system to a new 'current state'. The goal now is to reduce the solution space and with it, the number of system states and transitions to a useful size. This can be achieved by defining requirements that constrain the system in a way that particular states and transitions are no longer valid for the specified system. Such an approach limits solution space and a number of states with every additional requirement. Eventually, a meaningful, well-sized graph remains which represents the valid states and transitions for the specified system. Constraining the solution space and therefore reducing the system incrementally is a top-down approach.

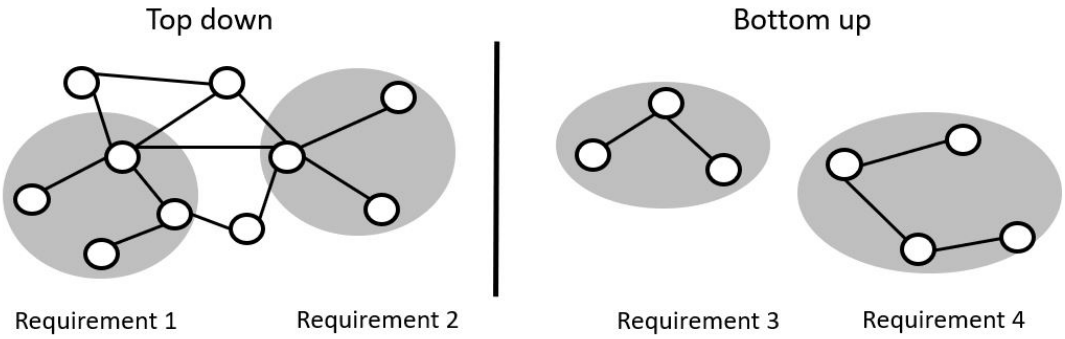


Figure 3.11.: System Specification Approaches: Top-Down and Bottom-Up

For this section, a top-down approach will be the one considered, while Section 4.3 uses a bottom-up variant for practical applications. The methodical implication for this description is, that requirements and tests affect states and transitions. Tests are commonly performed by using one path through the system and comparing the actual system states and transitions with the previously envisioned and defined state (and transition) description. Therefore, a test is represented in form of a forward-directed one-chain tree, as shown in Figure 3.12.

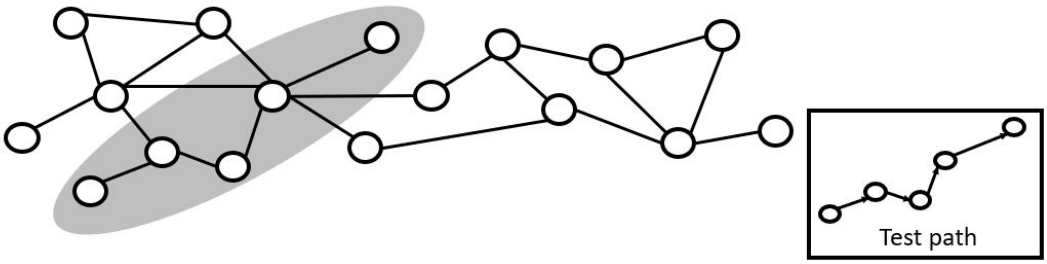


Figure 3.12.: System Graph Representation: Test

In contrast, requirements do not necessarily affect only one path through a system but a multitude of states and transitions. Figure 3.13 illustrates an example. The operating area of a requirement can include structures with forks (one state with two or more outgoing transitions), merges (one state with two or more incoming transitions) and loops (one state occurs two or more times as the 'current state' during a constraint description). Therefore, a more complex structure for descriptions is required - precisely a directed cyclic graph structure. This methodically caused structural difference between tests and requirements leads to a more complex processing for requirements data. It is specifically needed when LTL is mapped onto the structure to derive FOL.

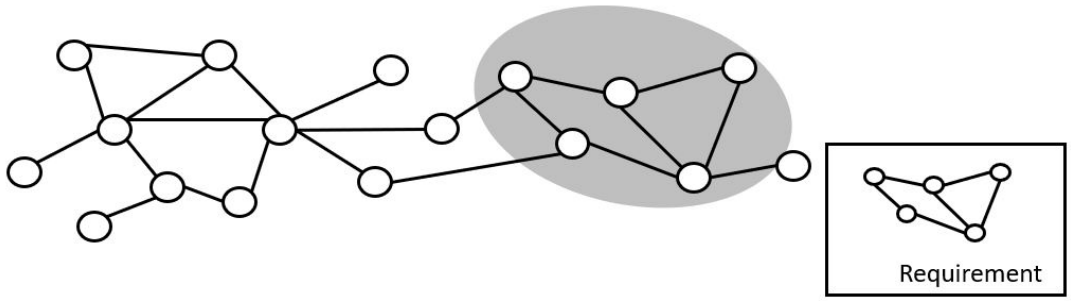


Figure 3.13.: System Graph Representation: Requirement

Because of the increased complexity for structures in combination with the logic expressions, this work only contains a case-based mapping for 'LTL to FOL' for directed cyclic graphs. A rule-based mapping is possible. Gastin et al. [GO01], Lu et al. [LL12] and others showed that for mapping of 'LTL to FOL' for Büchi automata, a complete rule set exists. MooreDFSMs, as defined in Subsection 2.3.3, are a special case of Büchi automata. The embedding theory "A general theory must formally include a special theory as a special case, else the special theory is falsified through the general theory or vice versa." [Rud02] can be applied. From there it can be concluded, that such a rule-based mapping must exist for 'LTL to FOL' for Moore DFSMs. However, this remains an open topic and will not be discussed and addressed further in this work.

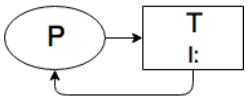
The case-based mapping is defined for all SPS patterns. The next section contains further discussion and qualitative examples while Section 4.3 addresses the industrial application.

3.5.2. Mapping Linear Temporal Logic on Directed Cyclic Graphs

To derive case-based mappings, a practical approach is to map the simplest occurring operators and elements and incrementally extend the mapping by adding additional logic operators. The four basic LTL operators are 'global' $\Box$, 'until' U, 'next' X and 'future' $\diamond$. Each operator is shown and explained in a basic mapping. These mappings can be combined for more complex structures. The mapping shows direct LTL to FSM mapping without explicit statements of FOL. The representation in FOL is an implicit representation in the states and transitions of the FSM. Each state and each transition is one particular FOL expression. For the basic operators it is trivial to represent FOL and for the complex operators it is not needed since these are aggregations of the basic operators. Therefore, no FOL is provided in the following mappings. Table 3.11 shows the operator

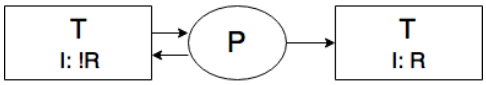
'global' $\Box$. There exists a state P with a transition into and out of this state. Because there is no further definition of the transition, it remains generic with no particular transition condition. To include 'P is true' in a system state machine, all states for the given system are extended with P. In addition, for each state (if not existent yet) an incoming and outgoing transition is added. This means it is generally possible to transition to and from the state but it is still unknown, what the exact transitions conditions are.

Table 3.11.: Conversion: SPS to LTL (Operator: 'Global' $\Box$), Walter et al. ([WHPR17])

Pattern	LTL
P is true globally	$\Box$ P
	

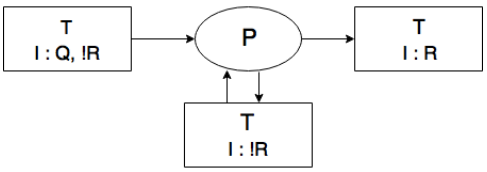
In Table 3.12, the example for the operator 'until' U is given. The state is expressed as 'P is true before R'. Therefore a state P exists. P remains until a transition with condition R occurs. As before, the state P can be reached by a generic transition. This condition can have all input conditions other than R, thus certainly $!R$. 'Current state' can cycle in and out of P until R is fulfilled. The transition with R equals true leads out of state P.

Table 3.12.: Conversion: SPS to LTL (Operator: 'Until' U), Walter et al. ([WHPR17])

Pattern	LTL
P is true before R	$\diamond R \longrightarrow (P\ U\ R)$
	

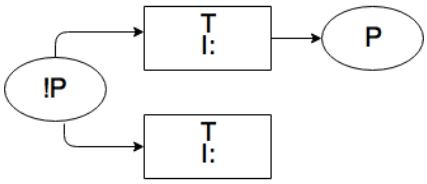
The example for operator 'next' $\circ$ is given in Table 3.13. 'P is true between Q and R'. The state P is reached when the input Q is given and R is not true (yet). From the next step after P is reached, the behavior is identical to that in the example given in Table 3.12. It can cycle in and out of the state P through the generic transition, and whenever R is true, it leaves the state P permanently.

Table 3.13.: Conversion: SPS to LTL (Operator: 'Next' X), Walter et al. ([WHPR17])

Pattern	LTL
P is true between Q and R	$\square\ ((Q \wedge \diamond\ R) \longrightarrow P\ U\ R)$
	

The last example shows the operator 'future' $\diamond$. Table 3.14 includes an example and representations. 'P becomes true' means that eventually a state P exists. This state P is reached by a transition which again is not specific and therefore contains no transition condition. For P becoming true, P must be untrue ($!P$) beforehand, therefore the previous state is $!P$. While eventually P becomes true, the state $!P$ might not directly transition to state P and therefore contains a generic transition condition.

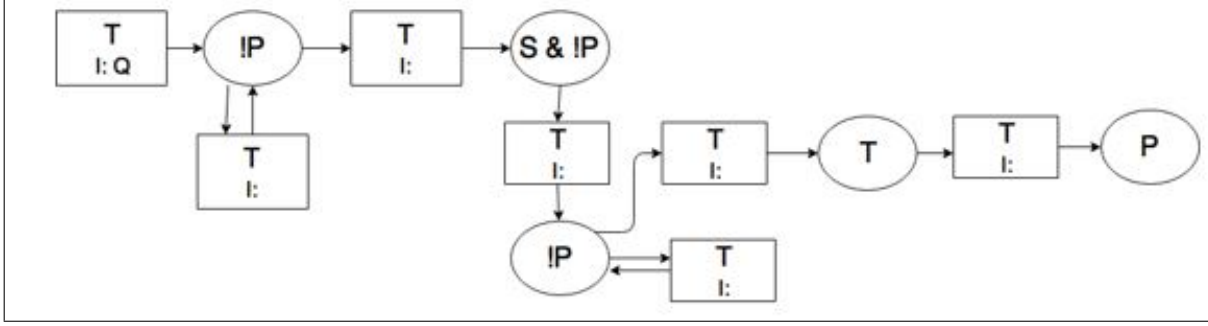
Table 3.14.: Conversion: SPS to LTL (Operator: 'Future' $\diamond$), Walter et al. ([WHPR17])

Pattern	LTL
P becomes true globally	$\diamond$ P
	

All these mappings show case-based solutions for single relations that occur within the SPS patterns. To illustrate the mapping with an applied example, Table 3.15 contains a concrete example with SPS, LTL and abstract FSM representation. Visualization of the concrete FSM can be achieved by replacing S with $LowBeamHeadlightLeft[On]$ AND $LowBeamHeadlightRight[On]$, P with $LightSwitchPos[Exterior]$ and Q with $Ignition[On]$ in the state machine. Since this would significantly reduce readability, it is just shown in abstract form.

Table 3.15.:
Conversion: SPS to LTL to FSM - Req. (Example), Walter et al. [WHPR17], adapted

	Pattern	LTL
Abstract	S responds to P after Q	$\square(Q \longrightarrow \square\ (P \longrightarrow \diamond\ S))$
Concrete	LowBeamHeadlightLeft[On] $\wedge$ LowBeamHeadlightRight[On] responds to LightSwitchPos[Exterior] after Ignition[On]	$\square(Ignition[On] \longrightarrow \square\ (LightSwitchPos[Exterior] \longrightarrow \diamond\ (LowBeamHeadlightLeft[On] \wedge LowBeamHeadlightRight[On]))))$



The last step for an unambiguous representation requires each expression at a state or transition to be shown in normal form. For a given FOL expression, this is a standard conversion. It is addressed with examples in the next section.

3.6. First Order Logic - Conjunctive Normal Form

Independent of the given modeling structure, each occurring state (node) contains a local expression. This expression consists of parameters (here: *literal*) and FOL operators (particular: $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NOT)). For a comparable representation, each expression must be converted in standard form. Bergmann et al. [BN13] characterizes such form as 'regular', 'easy to test' and with 'regular occurrence of quantifiers', all translated. Disjunctive Normal Form (DNF) and Conjunctive Normal Form (CNF) fulfill these characterizations.

$$CNF = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{d_i} (\neg) L_{ij} \tag{12}$$

L_{ij} in Equation 12 stands for any literal.

A DNF consists of literals which are connected through $\wedge$ operators (AND) $(A \wedge B)$. Each particular literal can be negated with $\neg$ (NOT) $(\neg A)$. A concatenation of literals is called a maxterm. Each maxterm contains any literal only once. Maxterms are connected through $\vee$ operators (OR) $((A \wedge B) \vee (C \wedge D))$. In contrast, a CNF connects literals through $\vee$ (OR) operators. These connections are again connected by $\wedge$ (AND) operators ($A \wedge (B \vee C)$). This is in analogy to maxterms for DNF and therefore shall be called *maxterm equivalent*. Each literal only occurs once in the overall conjunctive normal form expression. Therefore, CNF is more compact than DNF, which is the reason why from here on CNF is used as the standard form.

Equation 12 defines CNF formally. "For every formula that is free of quantifiers, one conjunctive normal form can be derived" [BN13], translated. 'Free of quantifiers' is in this case defined as 'only consisting of $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NOT)'. "Replace all occurrences of $\longrightarrow$ and $\longleftrightarrow$ with occurrences of $\neg$, $\vee$ and $\wedge$." [BN13], translated. Bergmann et al. further state that for a given CNF, its DNF can be derived and vice versa. "Dual to [the conjunctive normal form] is a type of normal form [...], which is called DNF." [BN13], translated. To illustrate the conversion to conjunctive normal form, a test step state description is converted in the below example from unprocessed FOL to DNF and CNF. To achieve CNF, a sequence of conversion steps must be performed against the statement.

1. Reduction of logic operators to $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NOT) This specifically means to remove operators $\longleftarrow$ (Implication) and $\longleftrightarrow$ (strong Implication)

2. Reduction of the affected area for negations. One negation $\neg$ can only affect a single literal)

3. Resolving combination connections of single literals towards maxterm equivalents. Single literals can only be connected to other literals through an alternation (OR) operator.

4. Connecting each maxterm equivalent through a connection (AND) operator. This leads to the form shown in Equation 12.

These four steps are applied to the expression shown in FOL. There exist no other logic operators than $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NOT), thus step 1 does not change the expression. Step 2 adjust the expression $\neg$ (TurnLightLeftFront[On] $\wedge$ TurnLightRight-Front[On]) towards two single statements with a negation each: $\neg$ TurnLightLeftFront[On]

$\wedge \neg$ TurnLightRightFront[On]. In step 3 the alternation on the top level is removed on the literal level: CountryCode[US] $\vee$ CountryCode[EU]. In the last step, all single literals, negated literals and the two literals with alternation are considered maxterm equivalents and are connected through combination operators. When comparing the derived expression for CNF with the equivalent DNF in the given example, it is obvious that CNF is significantly more compact. (DNF: nine literals; CNF: six literals). In addition to conversion into CNF, the resulting expression must be (when possible) sorted alphabetical to fulfill the 'regular' characteristics.

FOL (CountryCode[US] $\wedge$ ReverseLight[Off] $\wedge$ Gear[Reverse] $\wedge$
$\neg$ (TurnLightLeftFront[On] $\wedge$ TurnLightRightFront[On]))
$\vee$ (CountryCode[EU] $\wedge$ ReverseLight[Off] $\wedge$ Gear[Reverse] $\wedge$
$\neg$ (TurnLightLeftFront[On] $\wedge$ TurnLightRightFront[On]))

DNF (CountryCode[US] $\wedge$ ReverseLight[Off] $\wedge$ Gear[Reverse] $\wedge$
$\neg$ TurnLightLeftFront[On] $\wedge \neg$ TurnLightRightFront[On])) $\vee$
(CountryCode[EU] $\wedge$ ReverseLight[Off] $\wedge$ Gear[Reverse] $\wedge$
$\neg$ TurnLightLeftFront[On] $\wedge \neg$ TurnLightRightFront[On]))

CNF CountryCode[US] $\vee$ CountryCode[EU] $\wedge$ Gear[Reverse] $\wedge$ ReverseLight[Off] $\wedge$
$\neg$ TurnLightLeftFront[On] $\wedge \neg$ TurnLightRightFront[On]

The example data from Walter et al. [WHPR17] is processed in Table 3.16 in accordance with the conjunctive normal form and the alphabetical sorting. The first three lines of the initial example (1.1.1, 1.1.2 and 1.1.3) contain only single statements, therefore no sorting is needed. Further, all given examples do not contain logical structures that need adjustment to appear in CNF. It can be argued that therefore this example does not suit its purpose. The reason why the given example is still shown, lies in the bigger picture. Applying CNF in this context includes alphabetical sorting. Applying alphabetical sorting to the given examples, adjusts each particular statement. This is required for a fully unambiguous representation. When such a representation is achieved, the initial goal of the formalization process, namely: "[...] formalization of requirements and test data from textual representation to a formalized model form." given in Chapter 3, is fulfilled. The example from Table 3.16 therefore contains the final and most useful representation for each particular statement. To comprehend the overall process shown in Figure 3.1, conversion to CNF and alphabetical sorting represent the last and final step.

Table 3.16.:
Conversion: FOL to CNF - Test Case (Example), Walter et al. [WHPR17], adapted

	First Order Logic	Conjunctive Normal Form
...	...	...
1.2.1	(Gear[Neutral] ∨ Gear[Forward]) ∧ ReverseLight[Off] ∧ CountryCode[US]	CountryCode[US] ∧ (Gear[Forward] ∨ Gear[Neutral]) ∧ ReverseLight[Off]
1.2.2	CountryCode[US] ∧ Gear[Reverse]	CountryCode[US] ∧ Gear[Reverse]
1.2.3	CountryCode[US] ∧ Gear[Reverse]	CountryCode[US] ∧ Gear[Reverse]
1.3.1	IgnitionSwitch[Off] ∧ ReverseLight[Off] ∧ CountryCode[US] ∧ Gear[Reverse]	CountryCode[US] ∧ Gear[Reverse] ∧ IgnitionSwitch[Off] ∧ ReverseLight[Off]
1.3.2	(IgnitionSwitch[Lock] ∨ IgnitionSwitch[Radio]) ∧ CountryCode[US] ∧ Gear[Reverse]	CountryCode[US] ∧ Gear[Reverse] ∧ (IgnitionSwitch[Lock] ∨ IgnitionSwitch[Radio])
1.3.3	(IgnitionSwitch[Lock] ∨ (IgnitionSwitch[Radio]) ∧ ReverseLight[Off] ∧ CountryCode[US] ∧ Gear[Reverse]	CountryCode[US] ∧ Gear[Reverse] ∧ (IgnitionSwitch[Lock] ∨ (IgnitionSwitch[Radio]) ∧ ReverseLight[Off]

Throughout this chapter requirements documentation (including elicitation) and in particular all steps of the formalization process were shown. This included the documentation in NL with conversion to SPS or direct documentation in SPS. From SPS to LTL, the empirical mapping investigated by Dwyer et al. [DAC98] was shown. Walter et al. used the underlying modeling data structure to convert test data [WHPR17] and requirements data [WMR18] into FOL representation. To finalize the processing, all FOL expressions are put in CNF with alphabetical order, which means all data is represented in locally complete, unambiguous form. The derived data representation allows for further analysis and data optimization. Chapter 4 will apply the given formalization onto industrial data. This validates the formalization methodically by showing the impact and improvement in industrial setups.

4. Formalization Process Chain Applied to E/E Systems

"Pure logical thinking cannot yield us any knowledge of the empirical world; all knowledge of reality starts from experience and ends in it."

Albert Einstein, Ideas and Opinions

To solve a given engineering problem (in industrial context), an abstract approach (even if provable and correct) is only useful if it can be applied and adjusted to the given problem. This chapter applies the given formalization model of Chapter 3 to the industrial context. The correctness of each modeling step was previously shown. This allows the conclusion, that the results are generally correct on the syntactical level.

Table 4.1.: Chapter Structure - Publication Overview

	Formalization	Optimization
Testing	[WHPR17] - Section 4.1 Walter et al. - A Formalization Method to Process Structured Natural Language Requirements to Logic Expressions to Detect Redundant Specification and Test Statements	[WSPR18] - Section 4.2 Walter et al. - Test Cases Optimization through Removing Redundancies, Clustering and Reordering of Independent Test Steps
Requirements Engineering	[WMR18] - Section 4.3 Walter et al. - A Method to Automatically Derive the System State Machine from Structured Natural Language Requirements through Requirements Formalization	[WMS+19] - Section 4.4 Walter et al. - Executable State Machines Derived from Structured Textual Requirements: Connecting Requirements and Formal System Design

This chapter shows that the resulting data is meaningful and solves practical engineering problems. Four case studies are presented, two in the field of (system) testing and

two in the field of (system) requirements engineering. For both fields, one case study covers the general formalization while the follow-up case study presents an optimization, built on the initial formalization. Table 4.1 shows the classification of the four case studies, each with its primary publication. This leads to the overall structure of this chapter. For each case study, given application and research questions are laid out. The setup (system description and data set) is presented. Required processing methods are introduced before the application section shows the transformation (formalization or optimization) of the data set. Finally, achieved results are discussed and put in context.

4.1. Formalization of Test Cases

A common problem in the field of system testing is test load. The available time and resources require a reduction of all possible tests towards the most useful ones, or so to speak removing unnecessary tests. Unnecessary tests are tests which do not provide new information when executed. This is given when tests are redundant to each other or are a subset of the other test. For test cases represented in textual form, it is hard to evaluate (manual or machine-based) whether such a redundancy exists between multiple test cases. A data formalization can resolve the problem. This is shown with an industrial case study. The section is strongly based on the publication Walter et al. [WHPR17]. It cites the case study with its given data and generated results. The research questions raised in this context are paraphrased from that publication.

RQ I: Can the formalization be shown with a qualitative example?

RQ II: Can the formalization of textually represented test cases improve the detection of redundant test cases?

In order to answer these research questions, the assumptions and the industrial systems used are described in Subsection 4.1.1 and Subsection 4.1.2. A qualitative example is given in Subsection 4.1.3. Modeling and transformation are presented in Subsection 4.1.4 while Subsection 4.1.5 shows the resulting data set and closes with a discussion of these results. The term qualitative example refers to an example of a single test case with a step-by-step conversion. In contrast, quantitative examples will be used later as a term to show the approach in a scaled system.

4.1.1. Assumptions

The premise of this overall section is to answer the given questions **RQ I** and **RQ II**. Therefore, an industrial case study with two MBC systems is performed. The two systems are the OLC and the Intelligent Light System (ILS). Both systems were selected based on availability of the system data (tests and requirements) and willingness of the responsible engineers to participate in the research. Tests for both systems are initially given in textual form. Only test cases and test steps are considered for the later formalization. All data is stored in a requirements management tool that allows the export of data via Requirements Interchange Format (ReqIF). At Mercedes-Benz, the tool used for requirements engineering and test case descriptions is IBM Rational Doors. All data entries contain an attribute 'object type' with the value 'test case' or 'test step'. All other entries are excluded from this analysis. In addition, the father-child relation between particular test cases and test steps is known. Modeling of the data is performed in DC43. The model is generated with the information about 'object type', object relations and the processing descriptions given in Chapter 3. The test case description is shown in Figure 3.3 while the more detailed description for test steps is represented in Figure 3.4. This structure is exactly what is required for the intended processing. Data transformation was described for the generic data structure of a forward-directed chain in Figure 3.5. Therefore, the generic description must be aligned with the occurring data structure of the given test cases. This is shown in Figure 3.6. Data is reviewed regularly but not for redundancies. The given system data is described in the next subsection.

4.1.2. Setup

The two MBC systems investigated, can be described in the following way: "OLC controls turn and break signals, ILS adjusts outer lights based on outside brightness, other cars and pathway." In order to detect redundant tests, the given textual expressions shall be processed by the previously introduced process chain (see Figure 3.1). This includes a manual conversion from text to specification patterns and an automated conversion from these patterns to normal form. The representation in between specification patterns and normal form is in the form of LTL and FOL. The manual conversion from text to specification patterns is time consuming, which is why the analysis was limited to a reduced set of test cases ('random select'.) The selection was performed by choosing nine/seven (OLC/ILS) vehicle functions. *(A vehicle function is a car functionality which can be directly experienced by the user. Vehicle functions contain requirements which are*

linked to test cases.) These vehicle functions contained 64/137 (OLC/ILS) requirements and 35/52 test cases. 22/32 of these requirements have at least one test case linked while 43/103 requirements have no linked test cases. Requirements were initially not separated into testable and not-testable requirements.

Table 4.2.: Case Study: System Metrics (OLC/ILS), Walter et al. [WHPR17]

Object	Short	Total (OLC/ILS)		Random Select		Ratio[%]	
Vehicle Function	VF	400	389	9	7	2.25	1.80
Requirement	RE	3082	3731	64	137	2.07	3.67
Test Case	TC	1246	4228	35	52	2.81	1.23
Test Step	TS	4443	7120	233	136	5.24	1.91
Base Scenario	BS	21	25	6	8	28.57	32.00

Ratio equals number of 'random select' entries over total entries

"This leads to the seemingly high number of un-linked requirements which consist of non-testable requirements and requirements not tested on the system level (thus without links to system integration test cases)." [WHPR17] Table 4.2 shows both systems in full as well as the data set selected for the case study. This selected set is used for all further analysis. It is seen as representative for the overall systems and makes it possible to draw conclusions with general applicability. In the selection process ('random select'), a set of 9/7 vehicle functions were chosen randomly. All requirements and tests that were related to these functions were included as well.

4.1.3. Processing Methods

This subsection provides an example from the OLC system for step-by-step transformation. A test represented in NL is transformed into formal representation in CNF. This answers **RQ I**. The example was shown in Chapter 3 in detail. At first the example and the representation were shown in Walter et al. [WHPR17]. Table 4.3 shows the data for three test steps. Three columns (e.g. (1.1.1 - 1.1.3)) represent each one test step with precondition, action and postcondition. NL expressions are given initially. These are converted into specification patterns. This is the only manual step; all further steps are automated. For each pattern in SPS, a mapping in temporal logic is given and can be applied. These mappings are documented in Dwyer et al. [DAC98, DAC99]. Temporal logic can be mapped into state-wise FOL. This is described in Subsection 3.4.1 for a particular data structure (here called: 'directed one-branch trees').

Table 4.3.: Conversion: Full Process Chain - Test Step (Example)

	Natural Language	Specification Pattern System	Temporal Logic	First Order Logic	Conjunctive Normal Form
1.1.1	BaseScenario: Country Code is unknown	CCode[Unknown] is true at this state	CCode[Unknown]	CCode[Unknown]	CCode[Unknown]
1.1.2	Check Country Code for US	CCode[US]	CCode[US]	CCode[US]	CCode[US]
1.1.3	BaseScenario reached: Country Code is US	CCode[US] is true globally	G CCode[US]	CCode[US]	CCode[US]
1.2.1	Reverse gear and reverse lights are off	RGear[Off] AND RLight[Off] are true at this state	RGear[Off] AND RLight[Off]	CCode[US] AND RGear[Off] AND RLight[Off]	CCode[US] AND RGear[Off] AND RLight[Off]
1.2.2	Go into reverse gear	RGear[On]	RGear[On]	RGear[On]	RGear[On]
1.2.3	Reverse lights are off	RGear[On] is true before RGear[Off] AND RLight[Off] is true at this state	RGear[On] U RGear[Off] AND RLight[Off]	CCode[US] AND RGear[On] AND RLight[Off]	CCode[US] AND RGear[On] AND RLight[Off]
1.3.1	Ignition switch is off	IgnSwitch[Off] AND RLight[Off] are true at this state	IgnSwitch[Off] AND RLight[Off]	CCode[US] AND RGear[On] AND RLight[Off] AND IgnSwitch[Off]	CCode[US] AND IgnSwitch[Off] AND RGear[On] AND RLight[Off]
1.3.2	Turn ignition switch	IgnSwitch[Not Off]	IgnSwitch[Lock] OR IgnSwitch[Radio]	IgnSwitch[Lock] OR IgnSwitch[Radio]	IgnSwitch[Lock] OR IgnSwitch[Radio]
1.3.3	Reverse lights stay off	IgnSwitch[Lock] AND RLight[Off] are true at this state OR IgnSwitch[Radio] AND RLight[Off] are true at this state	(IgnSwitch[Lock] AND RLight[Off]) OR (IgnSwitch[Radio] AND RLight[Off])	CCode[US] AND RGear[On] AND ((IgnSwitch[Lock] AND RLight[Off]) OR (IgnSwitch[Radio] AND RLight[Off]))	CCode[US] AND (IgnSwitch[Lock] OR IgnSwitch[Radio]) AND RGear[On] AND RLight[Off]

CCode - CountryCode; RGear - ReverseGear; RLight - ReverseLight; IgnSwitch - IgnitionSwitch

Because test cases match this data structure, this case-based solution can be applied. This is the only step were information is transferred from one state (one line in Table 4.3) to another state. The resulting FOL is converted into CNF and sorted alphabetically. Tables 4.3 shows conversion of three steps and thus sufficiently answers **RQ 1**.

4.1.4. Application: Test Case Redundancy

To perform the analysis for test case redundancy, the tests have to be prepared (formalization) and processed (modeling). This only applies to the automated processing. Comments on the manual processing in regards to time effort are included in the next subsection. For automated processing, all textual descriptions are manually converted into specification patterns and exported via ReqIF from IBM Rational Doors to DC43.

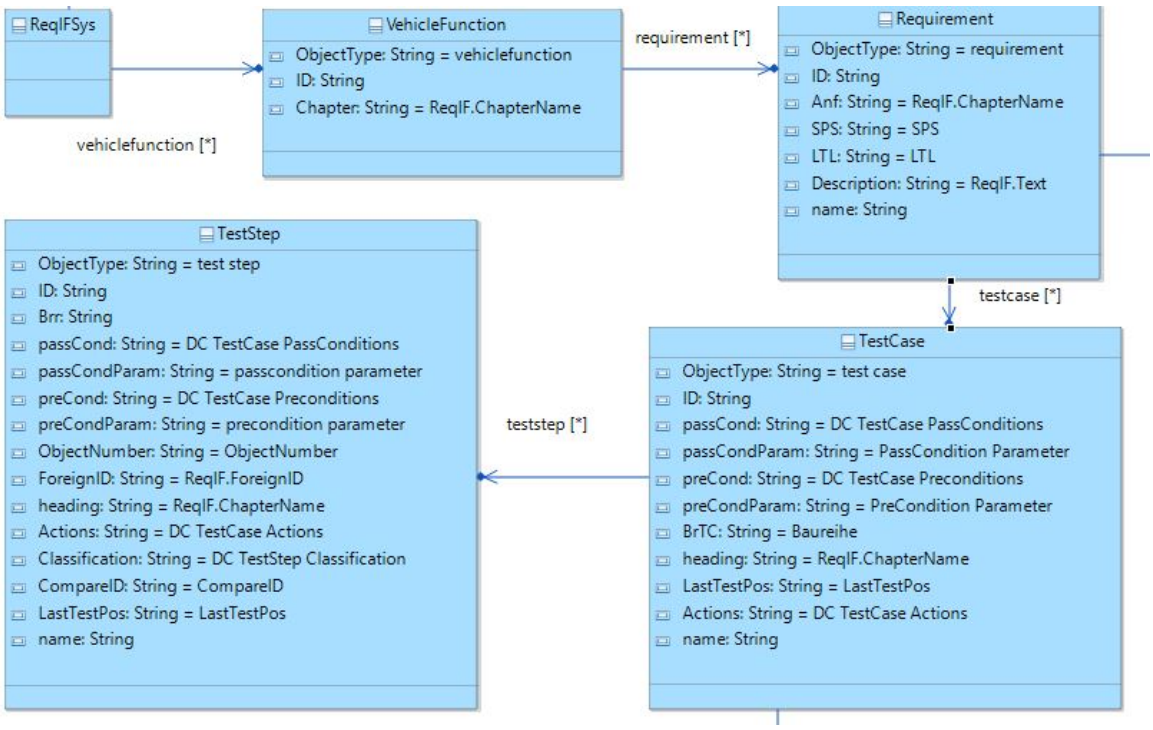


Figure 4.1.: System Representation: Class Diagram (Generic), (DC43)

The given data 'object type' and known relations are encoded in the class diagram, shown in Figure 4.1 in simplified form. It contains classes for system, vehicle function, requirement, test case and test step as well as their relations. The rule set is encoded in the execution diagram. This is shown in reduced form in Figure 4.2. The rule set contains the mathematical conversions for the formalization. Rules are executed sequentially.

Figure 4.2.: System Representation: Execution Diagram (Generic), (DC43)

A rule either instantiates objects from the given classes (here performed by the 'importer' and 'preprocessing' rules) or transforms objects (add or delete nodes and edges). Transformations are performed in rules 'SPStoLTL','LTLtoFOL' and 'FOLtoCNF'. Rule 'metrikFullData' generates some output metrics while 'compare' rule performs in accordance to its name, the matching of redundant test steps. Full representation of the system as instantiated objects in a graph is shown for the OLC sytems in Figure 4.3. Separation of classes in layers is performed for improved visibility.

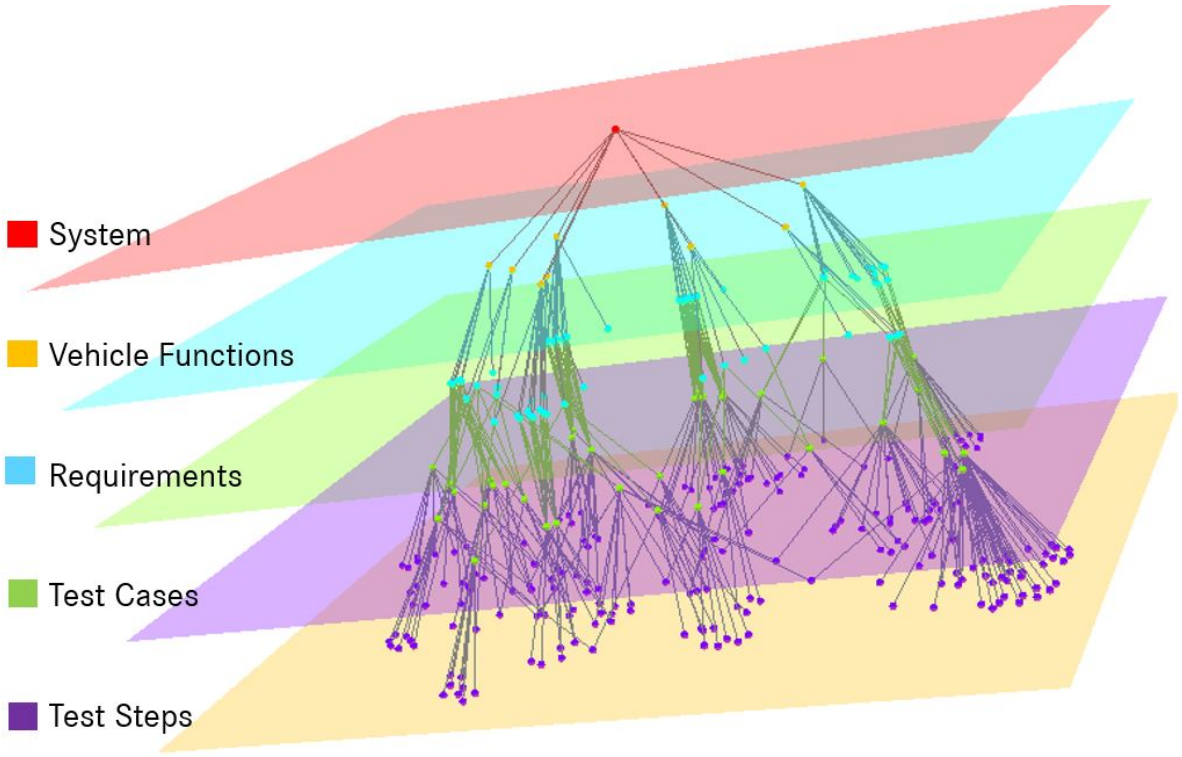


Figure 4.3.: System Representation: Design Graph (OLC), (DC43)

In regards to **RQ II**, the relevant analysis is, how well the machine-based approach performs compared to a manual redundancy check. The total number of findings and the required time effort are investigated. The reduced set given in Table 4.2 is used for all further analysis and seen as representative for the overall systems.

4.1.5. Evaluation

In this subsection, the raised research question **RQ II** is answered. The input for this is the given formalization and modeling performed in the previous section. The intention is to review the resulting data (test steps) in regards to detected redundancies and time effort for processing. Detection is separated into 'identical matches' and 'subset matches'. Identical matches occur when test steps contain the exactly identical description. Subsets differ in the sense that one test step fully includes the other test step description while adding further description (constraints). The more specific test step has to be maintained while the more general test step can be dropped. This is further elaborated on in Walter et al. [WHPR17]. Time effort is considered as a combination of effort for manual data formalization plus effort for manual redundancy checks. Walter et al. [WHPR17] writes: "[The system experts] reported that it took about two hours per system to define a parameter list for the statement descriptions. In addition 14/8 hours were needed to convert 233/136 test step descriptions to SPS (3.5 min per test step). This can be seen as the necessary additional work load required to prepare the system for further automated analysis". Walter et al. [WHPR17] further states: "[The] system experts performed an 8/5 hours manual review for redundancy of the 233/136 informal test steps." In comparison, the automated review was calculated in under five minutes.

Table 4.4.: Case Study: Review Findings (Manual/Automated), Walter et al. [WHPR17]

	Detection						Time effort [hours]			
	Identical (OLC/ILS)		Subset		Total		Formalization		Detection	
Manual	14	23	51	7	65	30	-	-	8	5
Automatic	16	23	54	25	70	48	16	10	5 min	5 min

The findings (as given in Walter et al. [WHPR17]) are the following: Results for 'identical matches' were almost the same for manual review and automated review (14/23 (manual) versus 16/23 (automated) findings). In contrast for 'subset matches', the manual review was outperformed by the automated review (51/7 (manual) versus 54/23 (automated) findings). Yet it must be remarked, that the difference in findings was only significant in the ILS, but not the OLC system. All findings were correct, but due to alternative representations and other factors it cannot be proven, that all redundancies were reliably detected. For the given system size, one manual overall system review takes about half the time of the complete data formalization. Therefore, it can be concluded that if the review is performed at least twice or the system size increases (n^2 comparisons versus

n formalization), no additional time invest is needed for the automated approach. This said, the results for matching indicate, that an automated approach is at least as good and in many cases better (+2/0 identical findings and +3/+18 subset findings) than the manual approach. The combination of formalization and redundancy check detects about 30% of statements (test steps) for an existing system as redundant. This alone might already justify such an approach in certain test setups. Dependent on the execution cost of a test, an additional advantage is the now existing formal representation of the system(s). Further automated data optimizations can be performed. This is shown with the example of test case restructuring as described in Walter et al. [WSPR18].

4.2. Post-processing of Formalized Test Cases

The possibility to detect redundant test steps through a formalization was previously shown. While this resolves the redundancies in tests, another problem has not yet been addressed: How can such redundant test steps be removed from an existing test set when they occur in a concatenation of test steps? This section addresses this problem. An industrial case study is provided along side with all needed processing steps. The major source for this section is the publication Walter et al. [WSPR18]. To discuss this problem, the following two research questions are raised:

RQ III: What steps, in addition to the given formalization, are necessary to rearrange test steps into a more efficient set of test cases?

RQ IV: Can the rearrangement of test steps in a set of test cases reduce the test load for a given system?

Both questions are based on the research questions provided in Walter et al. [WSPR18]. To answer these questions, the case study shown in Section 4.1 is continued. The underlying assumptions and additional information are provided in Subsection 4.2.1. Setup and input data set are presented in Subsection 4.2.2. The optimization includes some additional tasks which are not at the core of this work, which is why these shall be explained and shown with a qualitative example in Subsection 4.2.3. The application and actual processing is shown in Subsection 4.2.4, while Subsection 4.2.5 closes with a brief discussion of the results. All subsections are based on Walter et al. [WSPR18].

4.2.1. Assumptions

In order to answer **RQ III** and **RQ IV**, a case study is performed. The previously introduced MBC systems 'OLC' and 'ILS' are used. All premises introduced in the previous case study must be fulfilled in this context as well. Data must be presented in a requirements management tool with an export possibility to ReqIF. All objects must contain the attribute 'object type' with value 'requirement', 'test case' or 'test step'. Hierarchical relations between any objects and links between requirements and test cases must exist. All descriptions are initially given in textual form. Test step descriptions are converted manually from textual form to specification patterns. Further, it is assumed that: "The sequence of test steps does not carry a meta meaning, thus splitting and rearranging chains does not reduce information derived from executing all test steps." [WSPR18] In addition, it is assumed that information retrieved from the test set execution is constant for the 'initial' data set and 'optimized' data set. There is not intention to adjust this. The aim to optimize any execution 'effort'. In Walter et al. [WSPR18], 'effort' is named 'cost'.

Table 4.5.: State Space: Parameter Representation (Example), Walter et al. [WSPR18]

Parameter	Range	State 1	State 2	State 3	State 4
Par 1: *Time*	[0, 1000]	0	0	800	[200,500]
Par 2: *Velocity*	[0, 220]	180	0	0	[0, 50]
Par 3: *Lightswitch*	[Off, Stand, Day, On]	On	[Stand, Day]	Off	Off

To define 'cost', a state space shall be introduced: "The sum of all parameters occurring in at least one state defines the state space. Each parameter represents one dimension in this space as shown in Figure 4.4. This space contains all valid system states. All parameters listed in the description of a state must have either one specific value or a value range assigned to it (for example: $Par_1[Value_1]$). [...] A valid state therefore can be a specific geometric point, line, area or n-dimensional volume in the given state space." [WSPR18] The example shown in Table 4.5 and Figure 4.4 contains points, lines and areas. The case study in fact only includes precise points. The example is shown for states with three parameters and therefore three dimensions.

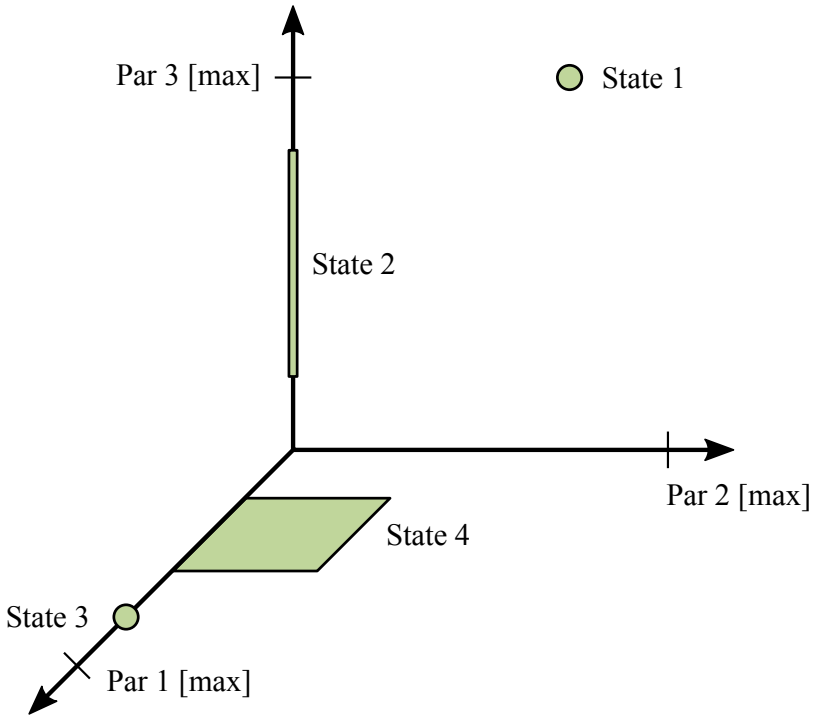


Figure 4.4.: State Space: Parameter Representation (Example), Walter et al. [WSPR18]

The state space is the basis for the cost metric. All costs are calculated through City-Block distance. Walter et al. [WSPR18] defines the cost metric in the following way:

1. **Initialization_and_Shutdown_Costs** - Costs to initialize and shutdown a test case
2. **Test_Step_Execution_Costs** - Costs required to execute one specific test step
3. **System_Change_Costs** - Costs to adjust system between consecutive test steps
4. **Total_Test_Case_Costs** - Sum off all costs for test step executions, system changes, initialization and shutdown of a test case.

   $C_{tc}(n) = C_{ini} + \sum_{i=1}^{n} C_{i,exe} + \sum_{i=1}^{n-1} C_{i,i+1,cha} + C_{shu}$
5. **Total_Test_Set_Costs** - Sum of all costs for test case executions of a test set.

   $[C_{ts}(m) = \sum_{j=1}^{m} C_{j,tc}]$

It is assumed that costs remain constant over time. This might not always be true (e.g. System_Change_Cost). Such costs might in reality follow a learning curve, yet this

simplification achieves to show the overall usefulness of the method and a potential small error would not affect the overall results and is therefore accepted. An example for the metric is shown in Figure 4.5. 'Cost' refers to the traveled distance in the state space from the start (initialization) of a test case to the end (shutdown).

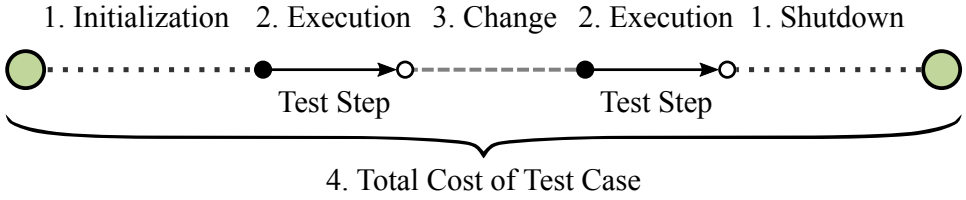


Figure 4.5.: State Space: Cost Metric - Test Case (Example), Walter et al. [WSPR18]

The overall premise of the optimization is to reduce cost while maintaining overall retrieved information from the test execution. The exact data set used from the case study is shown in the next subsection.

4.2.2. Setup

The case study is performed on the data set given for OLC and ILS in Table 4.2. For better visualization Table 4.6 shows the data specifically used for this application, as provided in Walter et al. [WSPR18].

Table 4.6.: Case Study: System Metrics II (OLC / ILS), Walter et al. [WHPR17]

Object	Total (OLC/ILS)		Random Select		Random Select Red. Removed	
Test Case	1246	4228	35	52	35	43
Test Step	4443	7120	233	160	181	126
$\frac{\text{TestSteps}}{\text{TestCases}}$	3.57	1.68	6.66	3.08	5.17	2.93

Total numbers for both systems are shown: 'random select' represents the data initially used in the case study, while 'random select - redundancy reduced' represents the formalized system after all redundant test steps are removed. In this context, 'random select' is used as the given data set while 'random select - redundancies removed' represents the optimized set of tests. The next section introduces clustering algorithms, similarity measures and path finding algorithms. It additionally explains these in an example.

4.2.3. Processing Methods

This subsection serves two purposes: Application sequence explanation with a processing example and introduction of three more processing methods needed for the application (clustering, similarity measures and path finding). The qualitative example addresses and answers **RQ III**. It was shown initially in Walter et al. [WSPR18].

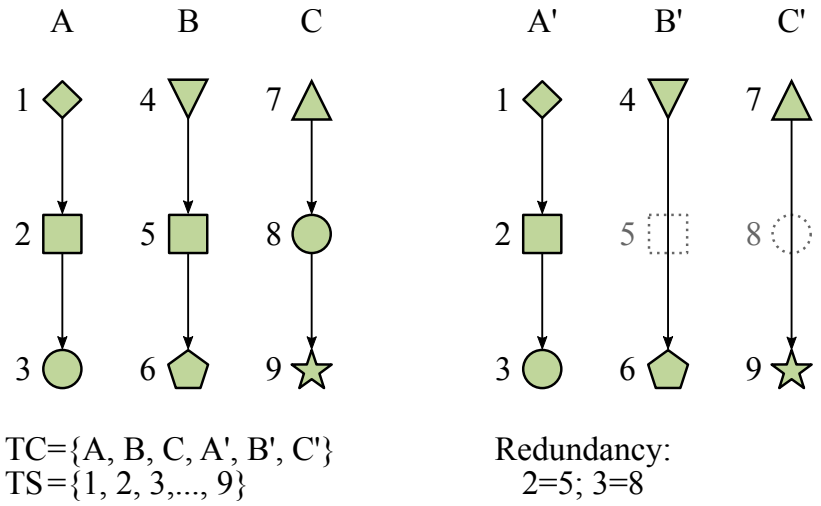


Figure 4.6.:
Test Set Reordering: Redundancy Detection (Example), Walter et al. [WSPR18]

In Figure 4.6, the initial set of three test cases (A, B, C) with three test steps each is given. Redundant test steps are marked in the set of test cases (A', B', C'). All remaining test steps are placed in a state space (see Figure 4.4). Walter et al. [WSPR18] considered three different algorithms to regroup test steps into test cases:

Centroid Based (K-Means): Clusters are formed by minimizing total distance of cluster points to cluster center of gravity. The point (pre- or postcondition), whichever is further away from the center of gravity, is used for determining that distance.

Density-Based Spatial Clustering (DBSCAN): Clusters are built through high density areas while borders emerge at areas of low density. Density is determined through the number of points in between a certain distance. In this case, 'distance' is defined as the shortest path between a given postcondition and any precondition.

Hierarchical Single-Linkage (SLINK): Clusters are generated by merging the two closest elements (points or arising clusters) into a new element (cluster). Here 'closest' is defined as the shortest distance between a given post- and any precondition.

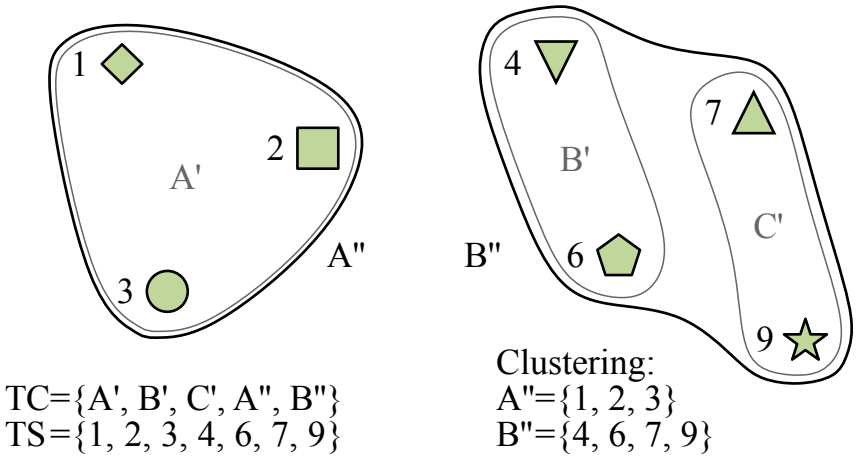


Figure 4.7.:
Test Set Reordering: Clustering Test Steps (Example), Walter et al. [WSPR18]

For each of the clustering algorithms, the number of clusters must be defined. K-Means and DBSCAN require an initial number of clusters to start while SLINK requires a goal number of clusters to stop. In this case study, the number of clusters for each algorithm is the number of test cases. Methodically each test case represents one cluster. A cluster is only removed once all its elements were moved to another cluster. To determine which elements are 'similar' to each other and therefore belong to the same cluster, three different similarity measures are considered by Walter et al. [WSPR18]:

Euclidean distance:

$$d_E(a,b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

City-block distance:

$$d_{CB}(a,b) = \sum_{i=1}^{n}|a_i - b_i|$$

Jaccard similarity (adapted):

$$sim_J(a,b) = \frac{\sum_{i=1}^{n} 1 - |b_i - a_i|}{\zeta + \eta}$$

Remark. *Similarity can be determined for City-Blocks and Euclidean distance based on coordinate distance for two points over all dimensions. a and b are two points in space. n is the number of total dimensions. For Jaccard ζ is the number of identical values and η the number of deviant values.*

Applying the clustering algorithm with the similarity measures against the initially given set of test cases (A', B',C'), a new set of test cases (A", B") with elements (test steps) is created. The number of clusters is constant unless a cluster has no more elements. The creation of the new arrangement of clusters can be seen in Figure 4.7. The derived test cases contain test steps without a particular execution order. To find an optimal execution order (lowest 'cost' as defined in Subsection 4.2.1), Walter et al. [WSPR18] considered three path finding algorithms:

Brute Force Method: For small sets of test steps, Brute Force is sufficient. It calculates all possible variants and therefore provides the optimal solution. For small sets ($n < 8$ elements), this can be calculated in satisfactory execution time. For sets with more than eight steps, execution time becomes inefficient. We observed test cases from the industrial case study exceed the eight steps, thus making brute force inefficient.

Genetic Algorithm: Initial sequences are crossbreed and potentially mutated. The result is compared against the initial sequences through a fitness function. In case of an improved sequence, the initial sequence is updated.

Ant Colony System: Initial sequences are compared. Favorable parts are reused and combined with random exploration (see: Dorigo [DG97]).

When applying the path finding algorithm to the data set, an (optimal) execution order is derived. Figure 4.8 shows a qualitative solution. The order for test cases A" remains 1, 2, 3. This seems logical since this set was not adjusted. The new test case B" has a test step execution order 4, 7, 6, 9 which is, compared to the initial execution of 4,5,6 and 7,8,9, optimal in regards to cost. The assumption for such a restructuring is that test steps are independent from each other and the execution order of test steps does not affect the results of each single test step. In case this assumption cannot be fulfilled (e.g. if the sequence of test steps 4,5,6 provides a meaningful result), the sequence can be maintained by substituting sequence 4,5,6 by 4*,6*. This maintains the combination of steps and sequence 4*,6* can be included as one end (4*) and start (6*) for connecting other steps. A questions asked for this approach is whether test cases are generally needed when test steps are independent from each other. By creating test cases with test steps instead of one set with all test steps, specific subsets of test steps can be chosen if it is not intended or possible to execute all tests. Therefore, test cases remain relevant since they serve as a structuring element for test steps.

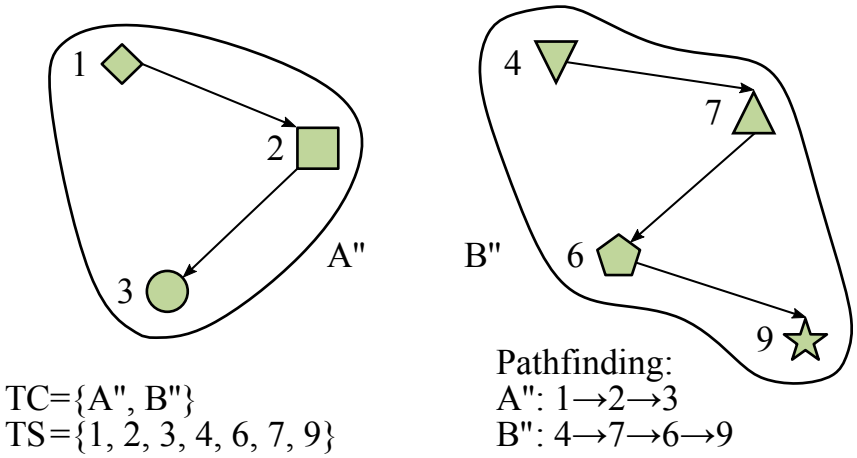


Figure 4.8.: Test Set Reordering: Test Step Sequence (Example), Walter et al. [WSPR18]

These processing steps combined lead to an overall set of test cases without any redundant test steps and a new execution sequence. Figure 4.9 represents the initial set (A, B, C) and the newly derived set (A", B") with test steps and execution order.

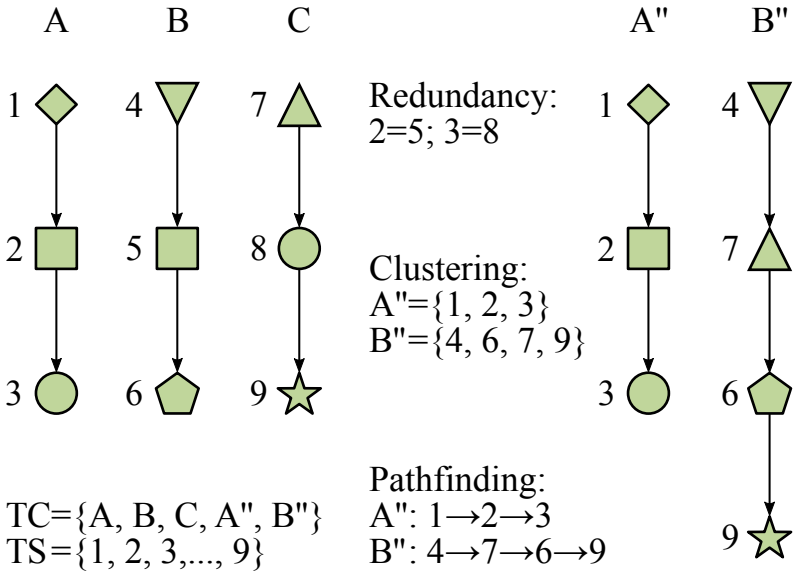


Figure 4.9.: Test Set Reordering: Overall Process, Walter et al. [WSPR18]

The next subsection explains the adjustments made in DC43 as well as the overall application for the case study.

4.2.4. Application: Test Set Restructuring

Data is imported via ReqIF from IBM Rational Doors to DC43. Data formalization is performed as shown in the previous section. Redundant test steps are removed and the minimal set remains as an input for test set optimization.

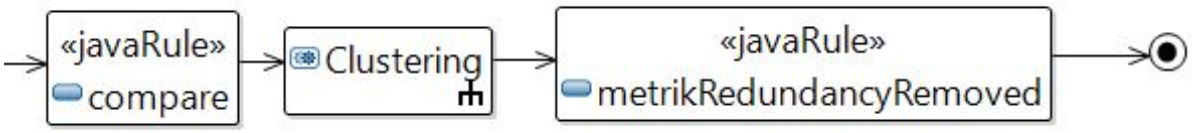


Figure 4.10.: Test Set Reordering: Execution Diagram (Extension), (DC43)

The class diagram remains the same as shown in Figure 4.1, while the rule set is extended by 'Clustering' and 'metrikRedundancyRemoved'. 'Clustering', includes the shown processing steps needed for the optimization. First the remaining test steps are sorted into new clusters. Second, the path finding is performed and the newly ordered test steps are put in an order. The 'metrikRedundancyRemoved' rule shows and analyzes the new retrieved data set. In regard to the design graph it can be said that no new objects are instantiated. The number of test step objects remains constant while the test case objects are reduced whenever a cluster (test case) is empty. The objects representing the empty clusters in the design graph are removed. Rearrangement of test steps is reflected in the design graph by adjusting the edges. The initial relations between test steps are removed and the new execution order is included in the graph through new edges between consecutive test steps. The optimization against the data set is achieved by applying the steps introduced in the previous subsection: formalization (removing redundancies), clustering (similarity measures) and clustering (path finding). In the next subsection, analysis is performed for all combinations of the three selected clustering methods and three similarity measures. All combinations are compared relative to each other and the best combination is used further. In a similar analysis, three selected path finding algorithms are compared relative to each other for improved test step order. All results are measured and compared through the previously defined cost metric. Finally, the total cost reduction for the applied method is shown for both systems. The results of the optimization are discussed.

4.2.5. Evaluation

The evaluation discusses results of the different clustering algorithms, similarity measures and path finding algorithms when applied against the data set. This answers **RQ IV**. All data is cited from Walter et al. [WSPR18]. In Table 4.7, the combinations for clustering and similarity measure are shown.

Table 4.7.: Case Study: Evaluation (Clustering + Similarity), Walter et al. [WSPR18]

	Euclidean	City Blocks	Jaccard
K-Means	99.6%	99.6%	100.0%
DBSCAN	87.3%	87.4%	90.4%
SLINK	89.6%	89.6%	87.5%

(All entries are represented in *Total_Costs* [%] relative to the *Total_Costs* of the **worst** combination.)

Walter et al. [WSPR18] concludes from this: "[...] all considered similarity measures provide almost identical results. [...] there is not one preferred similarity measure but preferred choices based on the selected clustering methods." The three best combinations are 'DBSCAN + Euclidian Distance', 'DBSCAN + City Blocks' and 'SLINK + Jaccard'. All three combinations achieve almost identical results.

Table 4.8.:
Case Study: Evaluation (Path Finding (min. & average)), Walter et al. [WSPR18]

Cluster size		Brute Force	Genetic Algorithm	Ant Colony System
'small cluster'	Ø	100.0%	117.2%	100.5%
(< 8 elements)	min.	100.0%	101.1%	100.2%
'big cluster'	Ø	*not possible*	119.7%	100.5%
(≈ 20 elements)	min.	*not possible*	101.2%	100.0%

(All entries are represented in *Total_Costs* [%] relative to the *Total_Costs* of the **best** combination.)

The size of clusters (test cases) cannot be controlled directly in DBSCAN. There exist parameters that impact the number of clusters indirectly but no explicit value for number of clusters can be set. SLINK can set an exact number of clusters. Since both algorithms (DBSCAN and SLINK) have a similar performance in regards to costs, SLINK is preferred since it allows direct controlling of the cluster size. The combination *SLINK, Jaccard* provides identical $Total_{Costs}$ results with the advantage of better controllability

of cluster size. With the combination 'SLINK + Jaccard', the path finding algorithm can be analyzed. Data is shown in Table 4.8 with a separation for 'small clusters (<8)' and 'big clusters'.

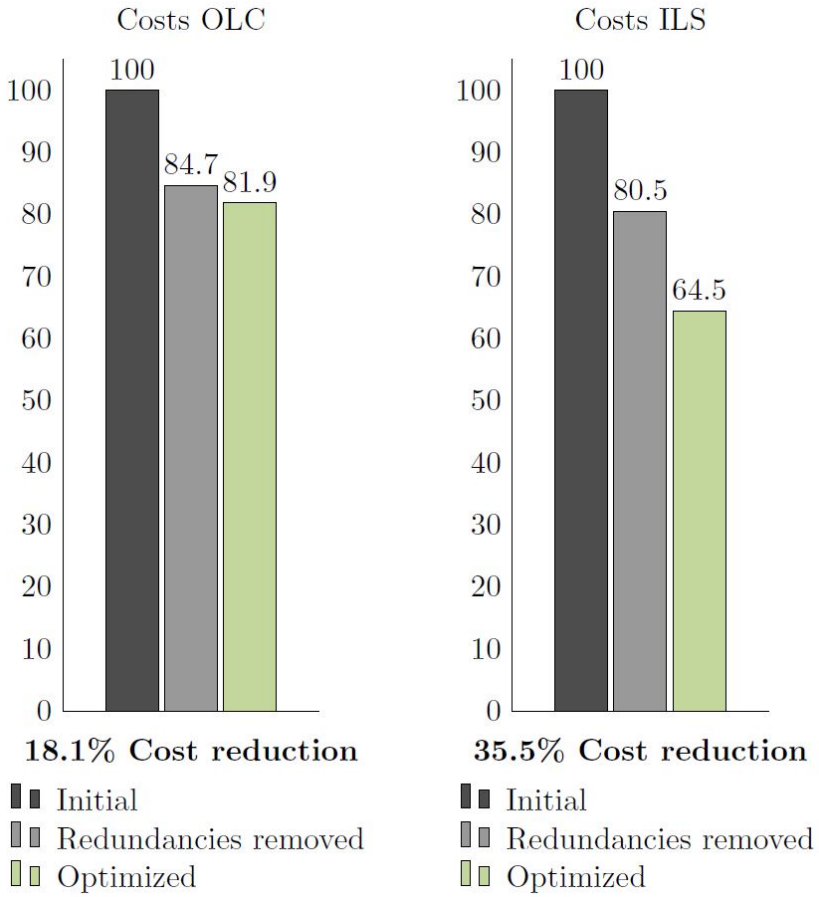


Figure 4.11.: Test Set Reordering: Evaluation - Overall Cost, Walter et al. [WSPR18]

"By considering all possible options, Brute Force provides optimal solutions when applicable. Because Brute Force only provides a reasonable run time for clusters with less than 8 elements, it should not be used on bigger clusters due to an enormous increase in runtime. [...] Genetic algorithm and Ant Colony System do not achieve optimal results since not all variants can be considered in a run. The advantage lies with the scalability capabilities for increased cluster sizes. Ant Colony System provides (almost) optimal results for 'small' clusters and is therefore an alternative to Brute Force. For 'big' clusters, the results are on average much better than the results of the Genetic Algorithm." [WSPR18] The conclusion is that for 'big' clusters, the Ant Colony System shall be used. For small clusters, this can be handled the same way or Brute Force can be ap-

plied. The selection of choice is 'SLINK + Jaccard + Ant Colony System'. Both systems are analyzed with this combination of algorithms, which is shown with overall results in Figure 4.11. "[It] shows a significant potential for removing redundant test steps [limited versus redundancies removed] (15.3% (OLC)/19.5% (ILS))." [WSPR18] The problem with the set of 'redundancies removed' test cases is, that it is not an executable set anymore. Test steps within the test cases were removed and therefore the execution chain in many test cases is broken. This is illustrated in Figure 4.6 with test step 4. Therefore, further reduction from 'redundancies removed' to 'improved' in Figure 4.11 is necessary. "The shown method of rearranging test steps into different test cases and changing the execution order reduces test execution costs by (2.8%/16.0%). Savings from removing redundant test steps (15.3%/19.5%) and reduction through rearrangement (2.8%/16.0%) add up to the total cost reduction (18.1%/35.5%). This total cost reduction is stated in Figure 4.11. In the author's opinion, the significant difference is caused by the more complex test step description for most of the ILS test steps. In addition, we see the discussed ratio $\frac{\text{TestSteps}}{\text{TestCases}}$ (see Table 4.6) as another reason. The ratio for the OLC (5.17) before clustering is much bigger than the ratio for the ILS (2.93) before clustering. Bigger test cases correlate with less execution costs. To answer [**RQ IV**], it can be said that both systems can be optimized (2.8%/16.0%). [...] Overall improvement is achieved at (18.1%/35%). Therefore, [it can be concluded], that [the] derived method is useful and can be applied (at least) to test data at MBC development." [WSPR18]. The combination of the two approaches in the previous section and this section provides a sufficient solution to the formalization and optimization of textually described tests. This was shown in analytical step-wise transformations and two quantitative case studies. The next two sections address formalization in the field requirements, paired with FSM.

4.3. Requirements Formalization - State Machines

Complex systems can often not sufficiently be described in textual form (alone), which is why model-based requirements engineering approaches become more relevant. One form of model-based requirements representation are FSM. Certain disciplines (marketing, law, general management, ...) often still require textual representations: A T2M transformation can bridge these two needs. M2T conversion is also possible but it requires initial specification in models. Because state of the art in the automotive industry is textual specification, T2M seems more relevant and suitable in this context. A text-to-logic conversion was shown in Section 4.1. This section introduces a logic-to-model

transformation. The combination achieves the proposed T2M conversion. To explain the logic-to-model conversion, two research questions shall be raised:

RQ V: Which steps are required to formally derive a state machine from textual requirements that describes the overall system?

RQ VI: Can the correctness of the derived 'system state machine' in regards to the initial textual representation be shown?

Both research questions are based on research questions given in Walter et al. [WMR18]. **RQ V** is limited in scope: Requirements formalization is already covered in **RQ II**, thus in **RQ V** only the transformation from logic representations to a system state machine is covered. The research questions are discussed in context of a case study with a MBC system: AOLC. AOLC is not related to the OLC system presented in the previous case studies. This section will first introduce all given assumptions for the case study, shown in Subsection 4.3.1. The setup for the case study is presented in Subsection 4.3.2. Generic state machines were defined in Subsection 2.3.3 and a general mapping from logic to state machines was previously introduced in Section 3.5. Yet the particular state machines used in this context still must be defined. These definitions are given alongside a qualitative example in Subsection 4.3.3. In Subsection 4.3.4, processing of the case study data is shown and explained. A discussion on the results and the conclusion is given in the evaluation in Subsection 4.3.5. This section is strongly based on the publication Walter et al. [WMR18].

4.3.1. Assumptions

This section aims to answer the raised research questions. **RQ V** is mainly addressed in Subsection 4.3.4 while **RQ VI** is discussed in Subsection 4.3.5. Both questions are answered with the help of a case study. The system under investigation is the MBC system AOLC. Data is provided in a requirement management tool with ReqIF export possibility. Objects are tagged with the attribute 'object type' (requirement, test case, ...). Relations are given through hierarchical order and linking between objects. All requirement descriptions are provided in SPS. *SPS to LTL mapping is performed within the case study while it is actually out of scope of the analysis since it was already discussed in the previous sections.* Test cases and test steps remain in textual description. These are not needed for formalization but to answer **RQ VI** with the validation in Subsection 4.3.5.

4.3.2. Setup

The case study is performed with data from the AOLC system. This data set was first published by Föcker et al.[FHDW15]. It was used due to availability. Table 4.9 shows the AOLC data set. The total set of requirements is reduced to functional requirements for this case study. This is due to the fact that non-functional requirements cannot always be represented in SPS. Therefore, it is not given that such requirements can be processed to LTL and FSMs. Data processing (LTL to Requirement Finite State Machine (Requirement FSM) and Requirement Finite State Machine (Requirement FSM) to System FSM) is explained in the next subsection. "[The] approach to validate the method is to validate the generated FSM through the given set of test cases and its relations (links) to the set of requirements." The assumption is, that if a manual match between the test case and state is found and the existing (indirect) linking between test case and state supports that same match, the state actually describes the requirement correctly." [WMR18] This is further discussed and analyzed in Subsection 4.3.5.

Table 4.9.: Case Study: System Metrics I (AOLC), Walter et al. [WMR18]

Function	Total Req.	Func. Req.	Test Cases
Turn and Warning Signaling	21	15	20
Low Beam Headlights	9	6	7
Adaptive (HB) Headlights	9	9	2
Manual (HB) Headlights	2	2	2
Fault Detection	6	3	6
Headlight Technology	3	3	1
Total System	50	38	38

4.3.3. Processing Methods

In this subsection, an example for all processing steps is shown. All transformation steps from logic representation towards representation in one system state machine are provided. "It is obvious that automotive systems require unambiguous transition to state relations. Therefore, all further discussions only consider deterministic state machines." [WMR18]. The extended definition of a Moore DFSM was given in Subsection 2.3.3. The short definition given in Walter et al. [WMR18] shall be recaptured:

Remark. ***MooreDFSM*** *(see full Definition 2.38): "Moore DFSM is a function with five variables": Moore DFSM = $f(I; TL; S; OL; O)$, where I = Input, TL = Transition Logic, S = State, OL = Output Logic, O = Output*

This generic Moore DFSM is used as the basis for three specific state machines which shall be defined in accordance to Walter et al. [WMR18]:

Definition 4.1. *"A* ***Requirement FSM****: represents exactly one requirement in the form of a state machine. Here a requirement stands for a requirement from the data set. Such a requirement is not necessarily atomic and might contain multiple pieces of specification information in it."*

Definition 4.2. *"****Atomic Requirement FSM****: represents one atomic requirement as a state machine. Here an atomic requirement either is equal to one requirement as defined above or, whenever requirements can be separated into multiple requirements, these resulting requirements shall be called atomic requirements. (For example, one requirements specifies a behavior for the left and right sides at once while this can be separated into two atomic requirements)."*

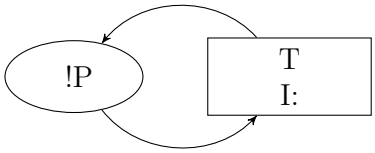
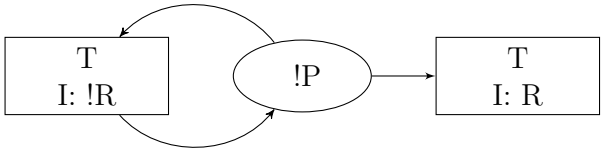
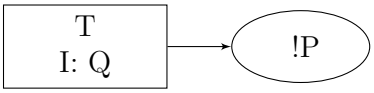
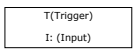
Definition 4.3. *"****System FSM****: represents unity of all requirements for the system."*

"A Requirement FSM represents exactly one requirement initially. It is possible that one requirement actually consists of multiple atomic requirements. Thus, a Requirement FSM is converted into one or potentially multiple atomic Requirement FSM. The break down into atomic requirements is necessary since the comparison and aggregation differs in case of atomic requirements (correct result) compared to 'pre-combined' requirements (not necessarily correct, (e.g. the detection of redundancies cannot be guaranteed in non-atomic representations)). Aggregation of all Atomic Requirement FSM build the System FSM." [WMR18] The first state machine in the transformation process is the Requirement FSM. Inputs are all logic expressions (concatenations of LTL and FOL expressions) that occur in Dwyer et al. [DAC98, DAC99] for SPS mappings. This is a limited set of 55 expressions (eleven patterns each with five cases). The mapping from logic to state machines for this approach can therefore be shown case by case.

It shall be noted here, that this work does not discuss LTL to FSM mapping in detail. A case-based approach is shown. This shall indicate, that the approach is generally correct and since a case-based approach exists, a generalized analytical solution is possible, yet this generalization is beyond the scope of this work. Works like Gastin and Oddoux

[GO01], Lu and Luo [LL12] and Villa et al. [VKBSV13] are referred to in order to obtain a generalized analytical solution. These references show the general possibility and correctness of such an approach but none of these references provide an applicable solution that can be directly implemented in these specific examples.

Table 4.10.: Conversion: SPS to Requirement FSM (Example), Walter et al. [WMR18]

Case	SPS	LTL
Globally	P is false	G (NOT P)
		
Before R	P is false before R	M (R) I (NOT P U R)
		
After Q	P is false after Q	G (Q I G (NOT P))
		
G - Global, X - neXt, U - Until, I - Implies, F - Future, ! - NOT	 	

"During 'LTL to FSM' mapping, the mapping from 'LTL to FOL' occurs. The solution for forward chains from Walter et al. [WR17] is adapted for this data structure. The divergent form of a state machine contains a finite number of forward chains. A forward chain is a single path through the state machine, where each state and transition occurs at most once. All possible unique chains are extracted. The given solution for forward chains is applied for each chain in isolation. The occurring LTL expression is converted to CNF and represented in a FSM." [WMR18] In Subsection 3.4.2, the mapping for 'one branch directed trees' was provided. This can be generalized towards 'directed cyclic graphs', which was shown in Subsection 3.5.2 and summarized in the above citation from Walter et al. [WMR18]. "Mapping from 'LTL to FSM' is mostly concerned with states and transitions. The reason for this is that Transition Logic and Output Logic are

simply functions that process a given input and generate a given output. The output only depends on the current state and does not feed information back into the system. In fact, Transition Logic, Output Logic and Output can be represented as attributes of a given state." [WMR18] Table 4.10 shows an example mapping for the 'P is false' pattern in three cases (globally, before R and after Q). Appendix B provides mappings for all SPS from 'LTL to FSM' with exception of 'chain response', which was never used in the case studies of this work.

Requirement state machines describe exactly one requirement in the form of a state machine. In general, a requirement shall be atomic in its description (see: ISO-29148 [ISO11]), yet there exist use cases where this principle can be at least questioned. For example, a symmetric behavior of left and right sides can certainly be specified in one requirement. To account for both situations (atomic and combined information), the separation of Requirement FSM and Atomic Finite State Machine (Atomic FSM) is made. In principle, all SPS are atomic, yet it is possible to combine multiple parameters into one substituted parameter. E.g. 'P is false' - '$(P_1 AND P_2)$ is false', where $P = P_1 AND P_2$. This allows requirement engineers to specify multiple pieces of information at once, while it is still possible to separate the requirements from one requirement state machine into two atomic state machines. More discussion on this is given in the next subsection (substitution - atomization library and substitute database).

Representation of a system in one state machine (System FSM) can be achieved by synthesis of all Atomic Requirement FSM. *In alignment with the earlier comment on generalized solutions, it shall be stated, that it is not intended to provide an optimized or generalized solution for state machine synthesis. This work rather shows the principle possibility with a selected non-optimal rule set, yet it solves the given problem. Further, existing work like Kam et al. [KVBSV12], Villa et al. [VKBSV13] and Lu and Luo [LL12] shall be mentioned in regards to generalized solutions.* These works prove that a general solution exists, yet no specific solution tailored to the particular problems in this work are given. Therefore, a rule set for prove of concept is provided. Synthesis of 'Atomic Requirement FSM' towards one 'System FSM' can be achieved by an alternation of a 'minimization process' and a 'generalization process' applied against the given set of 'Atomic Requirement FSM'. The intention is to minimize the occurring states and transitions in the 'System FSM'. Walter et al. [WMR18] described a rule set of three rules for minimization:

Rule Set - *Minimization*, from Walter et al. [WMR18]

(1) Merge Transitions

Two given transitions with identical start and target state and identical transition conditions (input) can be merged.

$T_1((S_1, S_2), I_1) = T_2((S_1, S_2), I_1) \rightarrow T_{12}((S_1, S_2), I_1)$

(2) Merge States

(a) Two given states with identical state descriptions *('descr.')* can be merged.

$S_1(descr_1) = S_2(descr_1) \rightarrow S_{12}(descr_1)$

(b) Two given states with different state descriptions but identical incoming and outgoing transitions can be merged.

$T_1((*, S_1), I_1) = T_2((*, S_2), I_1)$

$T_3((S_1, *), I_2) = T_4((S_2, *), I_2)$

$S_1(descr_1) \neq S_2(descr_2) \rightarrow S_{12}(descr_1 + descr_2)$

(3) Add Links

For two given states, where S_1 is a subset of S_2, all outgoing transitions from S_1 can be added to S_2.

$S_1 \subseteq S_2,\ T_1((S_1, S_3), I_1) \rightarrow T_2((S_2, S_3), I_1)$

The minimization connects states and transitions of different 'Atomic Requirement FSM' to create the one 'System FSM'. In the generalization, it is checked whether an 'Atomic Requirement FSM' contains global information. In case that this is given, this information is applied to all states and transitions in the 'generalization'. Walter et al. [WMR18] provides an example for generalization:
Example: S responds to P globally
'Recognition pattern': Find each occurrence of P
'Adapt FSM to': Add S to every consecutive state

"The synthesis process follows three overall goals. It minimizes occurring states and transitions, it connects related states and it shows potential inconsistencies through more exact specification of particular states and transitions. In sum, this synthesis combined with the previously described steps delivers the initially postulated System FSM." [WMR18]

The next subsection addresses the application where the overall process is performed against the data set of the case study.

4.3.4. Application: State Machine Representation

This subsection shows the data processing from the initially given data set in Table 4.9 to the data set derived in the consecutive subsection. It intends to answer **RQ V**. Figure 4.12 shows the execution diagram from DC43. Data is imported through ReqIF and pre-processed. 'SPS to LTL' is adjusted from the initial rule (see: Figure 4.2). This adjustment is related to the 'substitution' of parameters, which is explained when the overall process is discussed. 'LTL to FSM' creates Requirement FSM from the given LTL expressions. In the next rule 'CreateAndLinkASM', the Atomic Requirement FSM (in the rule: ASM) are derived from the Requirement FSM. The last task, 'synthesis', applies the previously described rules for 'minimization' and 'generalization' in alternation. This creates the System FSM. The described steps serve as a brief summary of the processing. A more detailed description is given now, alongside with the process overview.

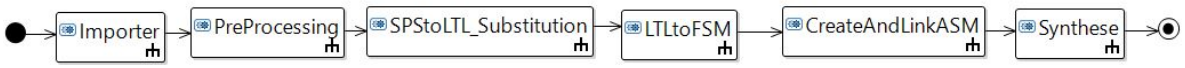


Figure 4.12.:
State Machine Generation: Execution Diagram (Extension), (DC43)

The overall process is shown in Figure 4.13. It can be separated into three groups of tasks and methods. The left side represents the 'core process'. Requirements are converted from NL representation to CNF form through the process steps introduced in Section 4.1. Subsection 4.3.3 discusses conversion from requirements in logic representation to a System FSM. The five methods in the center are libraries and transformation algorithms. The two mapping libraries 'SPS to LTL' and 'LTL to FOL' contain case-based mappings. 'Sorting algorithm' converts any FOL representation into a conjunctive normal form. Before 'atomization' is discussed, the remaining algorithm 'synthesis' is covered first. It has the 'minimization' and 'generalization' rules (see: Subsection 4.3.3) encoded and applies these in alternation until the System FSM is created and remains stable. 'Atomization' is the algorithm with most dependencies to other processing steps. Its inputs are the requirement state machine and information from multiple helper classes. The right side of the process overview contains supporting processes to mainly achieve atomization. 'Atomization library' (which could also be sorted into 'library and algorithms' group)

contains all SPS in atomic form. This allows a comparison of actually used patterns in 'requirement SPS' and generic patterns in 'atomic SPS'. Comparison is performed in 'atomization library'.

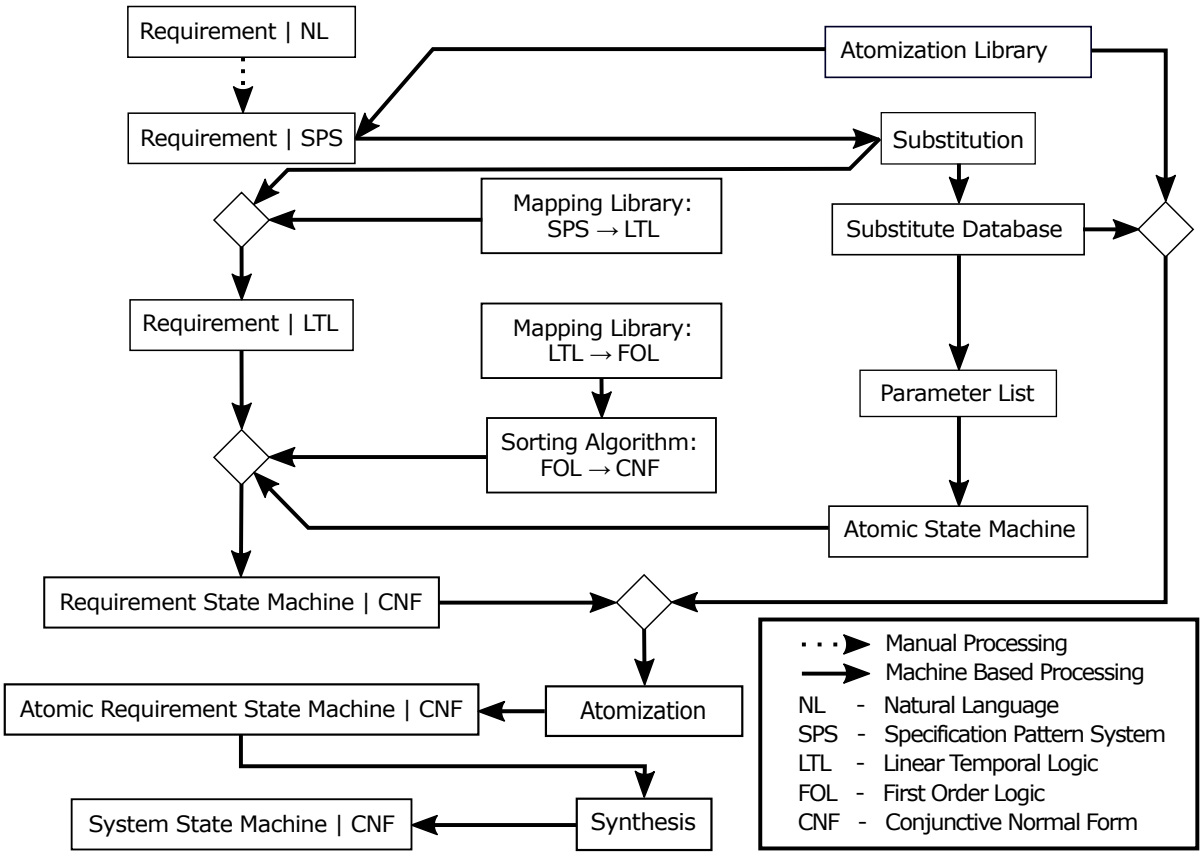


Figure 4.13.: State Machine Generation: Overall Process, Walter et al. [WMR18]

If any deviations occur, a substitution is made. Substitutions are calculated in 'substitutions' and stored in 'substitution database'. 'Atomization database' can be used for another task. It contains all parameters with all values that occur within the system. 'Parameter list' collects these parameters and uses these entries to generate 'atomic state machine' for each parameter. This aligns with the idea of graph-based design, where core building blocks are used to create more complex structures. For each parameter a state machine is created. This state machine contains all occurring values as states and allows all transitions between these states. This is called 'atomic state machine'. All complex state machines are combinations of 'atomic state machines'. This overall process is used to transform the data set. In the next subsection, a validation for the approach is shown.

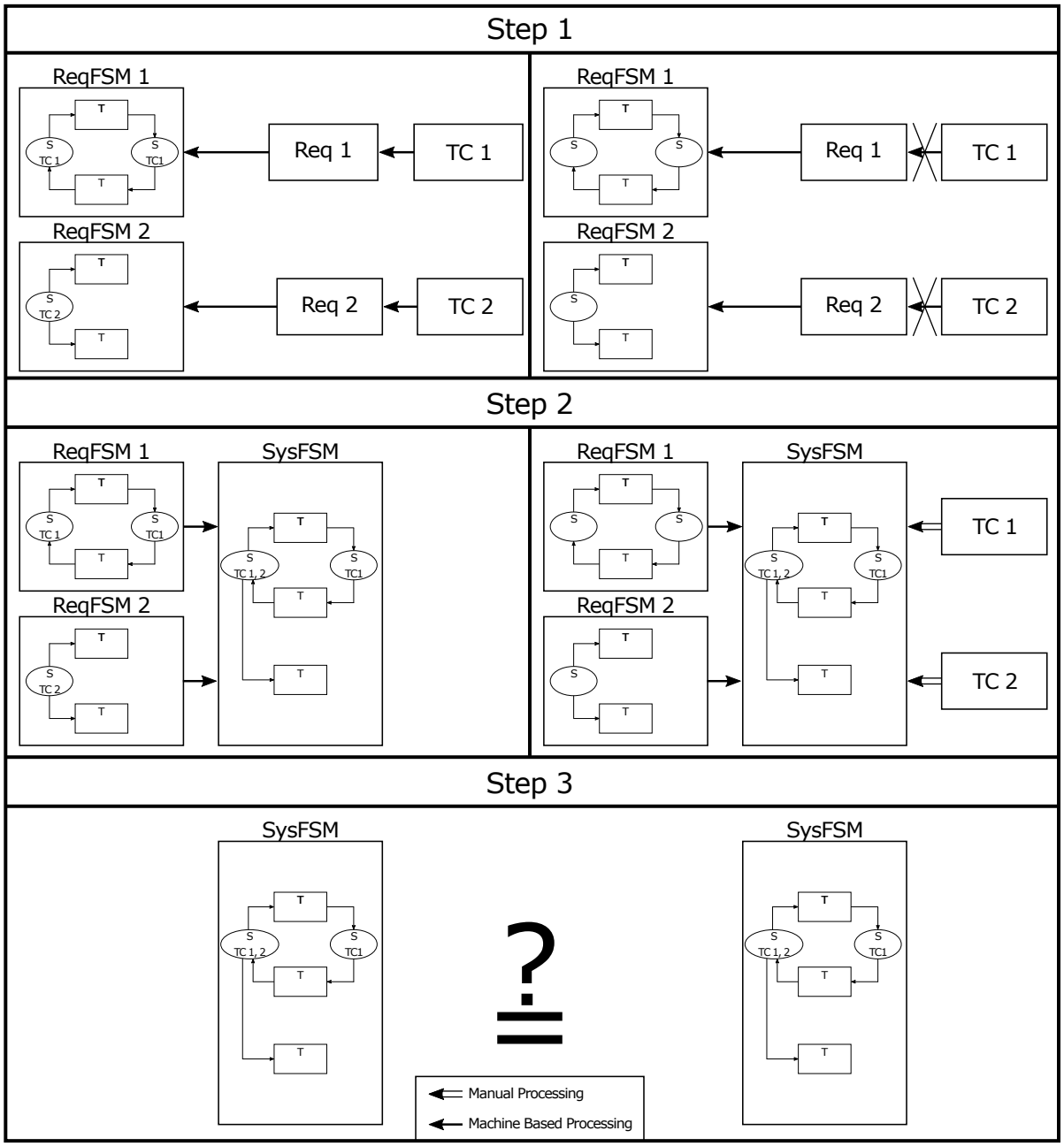


Figure 4.14.:
State Machine Validation: Comparison (Manual / Automatic), Walter et al. [WMR18]

4.3.5. Evaluation

This subsection shows resulting data and provides validation for this generated System FSM. It answers **RQ VI**. Validation is based on a comparison of existing and manually drawn links for the existing set of test cases. AOLC system data is given in Table 4.11.

The worst combination is represented with 100% and consequently the lowest number equals the lowest cost. Costs were calculated as described in Subsection 4.2.1. Costs for initialization, test step execution and system change are all based on the space distance. Initialization costs equal the distance from O to the start of test step 1. Execution costs are the the sum of the execution of each test step, where the distance from start to end for each test step is used as its execution cost. Change costs are the remaining distances between end of test step n and start of test step n_1. Shutdown costs equal the costs from the end of the last test step back to O.

Table 4.11.: Case Study: Evaluation I (System FSM), Walter et al. [WMR18]

Functions	Total States	Equal Links States	Add. Links States	Missing Links States
Turn and Warning Signaling	18	4*	14	0
Low Beam Headlights	5	0	5	0
Adaptive (HB) Headlights	7	0	7	0
Manual (HB) Headlights	4	1	3	0
Fault Detection	3	2	1	0
Headlight Technology	3	3	0	0
Reset	7	7	0	0
Total System	47	17	30	0

*S = States; Total = Number of States; * = two functional requirements without link to test cases; Equal = Case (1); Add = Case (2); Missing = Case (3)*

From the initial 38 requirements, a System FSM with 47 states was created. Table 4.11 assigns each state to one particular function. This is done in order to avoid double consideration of a state in the analysis. From a functional standpoint, a state is usually not limited to serve only one function. The analysis is performed as shown in Figure 4.14. Two System FSMs of the AOLC system are created. On the left side (FSM_1), the requirements contain links to the initially given test cases. This provides a trace from state to requirement to test case. On the right side (FSM_2), the links between requirements and tests are removed. The created System FSM (FSM_2) is now given to a system expert alongside the set of test cases. The system expert links the test cases directly to states in the System FSM (FSM_2). The system experts were asked to only link test cases to states that definitely validate that state. The resulting links can now be compared for both System FSM (FSM_1 and FSM_2). Walter et al. [WMR18] discusses the classes of resulting links:

Case (I) - Equal Links: $IDs_{auto} = IDs_{man}$

It has to be checked whether all elements for IDs_{auto} and IDs_{man} are identical. If number of elements for IDs_{auto} and IDs_{man} are equal but differ in actual IDs, a combination of case (II) and case (III) occurred.

Case (II) - Additional Links: $IDs_{auto} < IDs_{man}$

Manual review linked more test cases to the particular states than the automation. It must be checked whether all IDs_{auto} are included in the list of IDs_{man}. In case any element of IDs_{auto} is not included in the list of IDs_{man}, a special case of case (III) occurs.

Case (III) - Missing Links: $IDs_{auto} > IDs_{man}$

Manual review failed to link at least one test case to a particular state that was linked through automation.

The results from Table 4.11 shall now be discussed. The automated approach assigned links based on the initially given structure without considering the content of the state. Links from test to requirement are considered correct since these were used in the industrial projects. These links are inherited from the requirements to the newly generated states. Therefore, a link from test case to requirement to state is considered correct. What shall be proven is that the content of the state actually matches the content of the requirement. Therefore, without considering the content of the requirement, the system expert directly links test cases to the created states. This proves that the content of the state matches the content of the test case. Since test case and requirement are initially correctly connected, this now makes it possible to check whether the newly created link (manual) and the content-based existing link match. This would prove content consistency between the requirement and the derived state. For states that have an equal number of automated and manual links, it must be checked whether these links are actually the same links (same test case to same state). This is the case for all 17 occurrences. Therefore, it can be concluded that these states are correctly derived from their requirements. In the second case, where manual links were exceeded by automated links, this same check has to be performed: Are all automated links also links that were drawn manually? The check is positive. Therefore, all states can also be considered correctly derived. The additional links can be seen as an improvement of the test set.

It is common that a test is specified for a particular requirement (or here a state) but it is not crosschecked whether another requirement (or state) can also be tested with this test case. Thus, it is not surprising that many states had additional tests linked when this was checked for in particular. There were no states with missing links, which further confirms the approach. Overall, it can be concluded, that the System FSM was successfully derived and validated through the described approach. The next section will discuss how this System FSM can be adjusted to be dynamic and controllable. In addition, a qualitative example of a requirement transformation from NL to 'System FSM' is shown.

4.4. Requirement Models - Executable State Machines

A formal requirements representation can explicitly express dependencies, prove consistency and highlight redundancies between requirements. Another relevant piece of information is the effect of a (new) requirement onto a given system. Often a requirement is specified but the exact constraints and implications on the system are unknown or at least not fully clear. Engineers can judge a specified requirement much better when its effect is observable in the system. An application that provides such a possibility is a dynamic or executable System FSM controlled by the requirements engineer. Dynamic and executable shall be used interchangeably within this context. This section introduces the remaining steps to adjust the derived System FSM in a way that it is controllable through inputs and behavior can be dynamically visualized and observed. Therefore, the following research questions are presented:

RQ VII: What steps are necessary to make automatically generated System FSM controllable through user input?

RQ VIII: Can the correctness of the conversion steps from static to derived dynamic System FSM be shown?

Both research questions are strongly based on Walter et al. [WMS+19]. Both questions shall be assessed with a case study. Subsection 4.4.1 shows the assumptions given for the case study, while setup and data set are introduced in Subsection 4.4.2. Subsection 4.4.3 introduces all additional methods needed. In addition, it contains a qualitative example of a requirement transformed from NL representation to System FSM. This is the basis on which the execution layer for the dynamic System FSM can be applied. Data processing

is shown in Subsection 4.4.4. This includes adjustments in DC43 and 'eTrice', an Eclipse extension to model dynamic state machines. The validation of the derived System FSM is performed in Subsection 4.4.5, followed by a brief conclusion. This section is strongly based on Walter et al. [WMS+19].

4.4.1. Assumptions

The section intends to answer **RQ VII** and **RQ VIII**. Therefore, an example and a case study are performed in Subsection 4.4.4. The system used for the case study is the MBC system AOLC. This system, with the same data set was used in the previous section to derive System FSM from requirements in logic expressions. All data is stored in a requirements management tool with a ReqIF export function. Objects must contain an attribute 'object type', which classifies the object into 'requirement', 'test case' and other types. Dependencies are given by hierarchy and links between objects. The scope of this section is only on adjusting the process in order to improve a given (static) System FSM towards a dynamic System FSM. Therefore, it is assumed that the 'requirement to FSM' conversion is given and the System FSM for the AOLC already exists. Requirements and tests are still needed for the validation in Subsection 4.4.5.

Table 4.12.: Case Study: System Metrics II (AOLC), Table 4.9 + Table 4.11

Function	States	Functional Requirements	Test Cases
Turn and Warning Signaling	18	15	20
Low Beam Headlights	5	6	7
Adaptive (HB) Headlights	7	9	2
Manual (HB) Headlights	4	2	2
Fault Detection	3	3	6
Headlight Technology	3	3	1
Total System	40*	38	38

** Total number of states differs to Table 4.9 because reset states are excluded in this representation. Reset states have to be included in the state machines, for a complete representation but do not affect this analysis (no description on reset states) and are therefore excluded for a better overview of the states relevant for the analysis.*

4.4.2. Setup

The data set used for the case study is the AOLC system. To create a dynamic behavior for the System FSM, the initially given data from Table 4.9 (requirements and test cases) and Table 4.11 (states) are combined for better visibility. This is shown in Table 4.12. The formalized requirements serve as the basis for the generation of the System FSM. This FSM contains the listed states. The intention is now, to create a layer that makes it possible to control the System FSM. This is described in the next subsection. The validation of such a layer can be performed by testing the generated System FSM against functional tests. "Linking between tests and requirement in addition to the traceability between requirement and system states make it possible to locate the system states related to a particular test case. Walter et al. [WMR18] showed correctness and practicality of the generated static System FSMs. We intend to verify the executable machine by execution of the related buttons at the Graphical User Interface (GUI) and comparing output to specified tests in a black box test. The assumption is that the requirements and test cases are correct. We do not intend to test the system specification but we show that the executable machine represents the specification." [WMS+19] This is shown in Subsection 4.4.5.

4.4.3. Processing Methods

In this subsection, an example for the formalization of a requirement from NL to System FSM is given. This answers **RQ VII**. It serves as a preparation for the execution layer that is added for controllability. Additional methods introduced to control the state machine are therefore explained in the next subsection to balance the subsections.

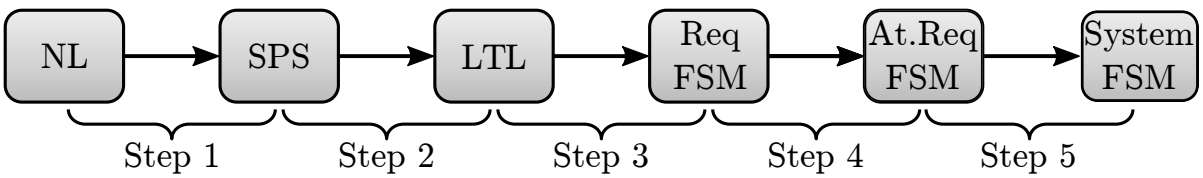
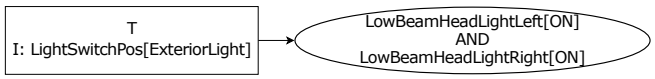
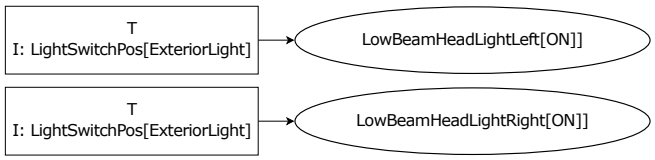
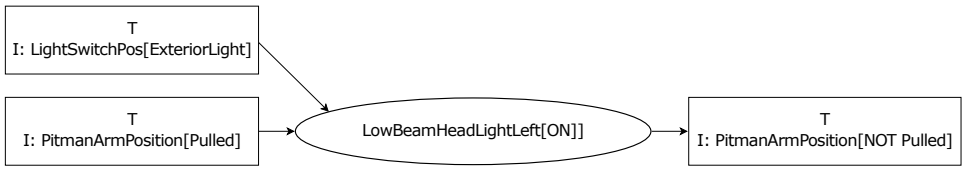


Figure 4.15.: Formalization Process Chain: NL to System FSM, Walter et al. [WMR18]

The example provided was given in Walter et al. [WMS+19] and is shown in Table 4.13. The mapping is 'SPS to LTL to FSM'. It is not bijective since multiple SPS patterns can generate the same LTL expression and multiple LTL expressions can lead to the same FSM. The mappings are further discussed in Appendix A. To make it easier to follow the

explanation in regard to the conversion steps, the processing chain shown in Figure 3.1 is extended and provided with step labels in Figure 4.15.

Table 4.13.: Conversion: SPS to System FSM (Example), Walter et al. [WMR18]

Representation Form	Requirement Representation
NL	Low beam headlight left and right are activated by turning light switch to position exterior lights
SPS	LowBeamHeadlightLeft[on] AND LowBeamHeadlightRight[on] is true after LightSwitchPos[ExteriorLight]
LTL	G (LightSwitchPos[ExteriorLight] I G (LowBeamHeadlightLeft[on] AND LowBeamHeadlightRight[on]))
Requirement FSM	
Atomic Requirement FSM	
Pairing FSM*	
System FSM**	
G - Globally, I - Implies	T(Trigger) I: (Input) state

** Pairing FSM is an additional FSM that is introduced in the example to illustrate the generation of the System FSM*
*** For simplicity, System FSM representation is limited to LowBeamHeadlightLeft[on]. Therefore, no LowBeamHeadlightRight[on] is shown in the System FSM.*

From the AOLC system data set, a requirement represented in NL is selected. The semantical meaning has to be understood and an appropriate SPS pattern is selected in *step 1.* "SPS should be represented with parameter names with syntax $Par_1[Value_1]$ e.g.

LowBeamHeadlightLeft[on]. Parameters in this example are: *LightSwitchPos*, *LowBeamHeadlightLeft* and *LowBeamHeadlightRight*. The selected pattern is 'Universality - P is true after Q'." [WMR18] In *step 2*, the specific parameters are inserted for P and Q. P is here substituted in $P_1 AND P_2$ to include both parameters *LowBeamHeadlightLeft* and *LowBeamHeadlightRight*. The substitution is stored in the substitution database. This representation can now be transformed into LTL, in accordance to the mapping in Dwyer et al. [DAC98, DAC99]. For the conversion into 'Requirement FSM', the mapping 'LTL to FSM' is used. Figure B.3 in Appendix B shows the particular conversion for the SPS 'Universality - After Q' needed for *step 3*. "In this case, *LightSwitchPos[ExteriorLight]* serves as the input (trigger) for the transition which causes both *LowBeamHeadlightLeft* and *LowBeamHeadlightRight* to turn from *'off'* to *'on'*." [WMR18] The specification initially combined behavior from *'left'* and *'right'* into one requirement.

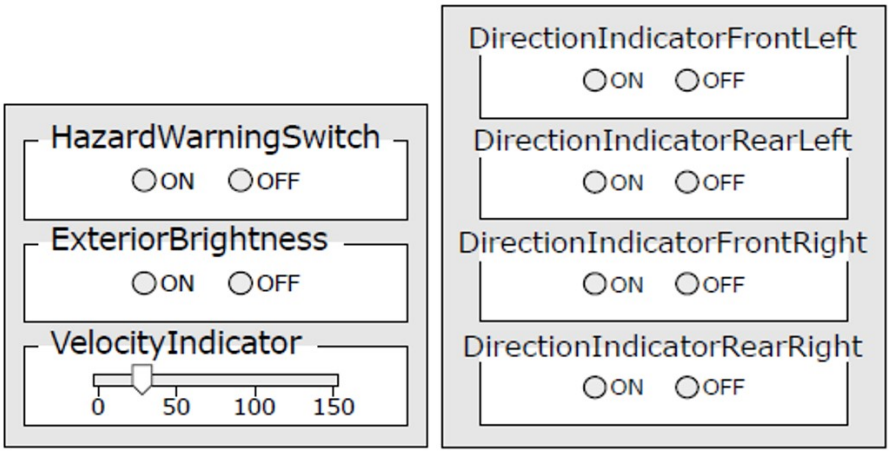


Figure 4.16.:
Dynamic State Machine Generation: Input / Output Layer, Walter et al. [WMS+19]

This is separated from one 'Requirement FSM' into two 'Atomic Requirement FSM' in *step 4*. Substitution database is accessed to provide the separation data. Remaining is the system synthesis in *step 5*. To illustrate a synthesis, a pairing requirement in the form of another 'Atomic Requirement FSM' is introduced. The given synthesis rules are applied. "[Identical] states can be merged. All incoming and outgoing transitions for either one of these states are added as well. This leads to one state with two incoming and one outgoing transition." [WMR18]. Overall, this shows the transformation from requirements in natural language to requirements as part of a System FSM. It is the basis for the dynamic System FSM. The execution layer is added in the next subsection.

4.4.4. Application: C-Code Generation

This subsection explains the necessary adjustments for the execution layer. It first introduces additional tasks and methods while later showing the adjustments at DC43 and the modeling in 'eTrice'. The dynamic behavior is encoded into the System FSM by introducing an 'execution layer'. This layer consists of "[an] external interaction layer with a GUI and output console as well as an internal processing layer. This layer contains the underlying logic and internal signal transfers. The FSM is exported to 'eTrice' and both layers are added to the existing System FSM." [WMS+19]. The external interaction layer is shown with its input/output GUI in Figure 4.16.

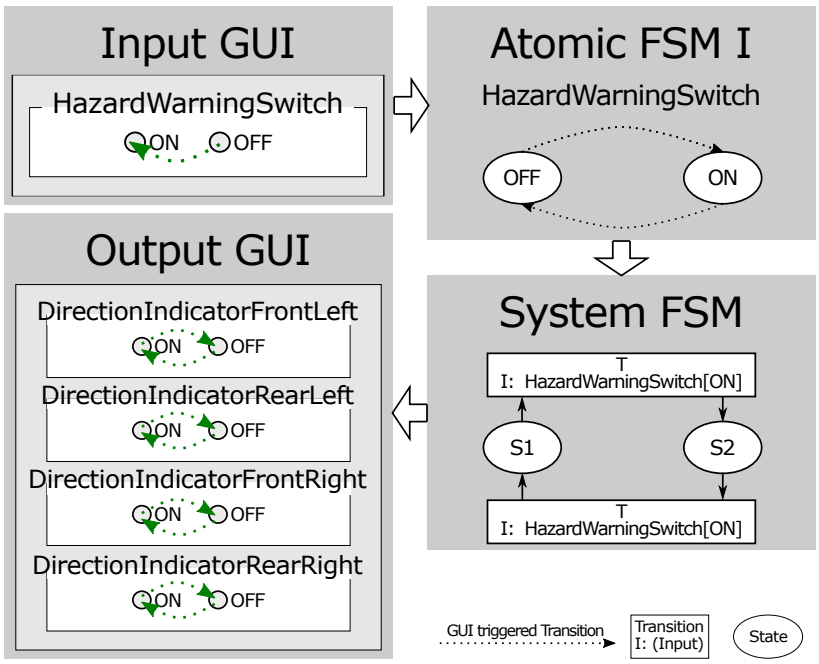


Figure 4.17.:
Dynamic State Machine Generation: Overview (Example), Walter et al. [WMS+19]

This layer allows user input to the System FSM and returns system output. "The parameters displayed at the GUI, are all parameters that affect the system to transition between states. The complete list of GUI entries is created by crawling all transition inputs for unique Parameter[StateValue] combinations. [...] Through internal logic, current state is transitioned to a new current state and a signal with an output message is sent. This output has to be represented alongside the new current state. Therefore, besides input control, the second purpose of the external layer is displaying output

messages." [WMS+19] The parameters occurring at the input are transition parameters while the parameters at the output are state parameters. The connection of System FSM with the 'external interaction layer' is achieved with an 'internal interaction layer'. The system state is represented in 'Atomic FSM'. Walter et al. [WMS+19] defined 'Atomic FSM' in the following way:

Definition 4.4. *"**Atomic FSM**: represents an FSM for one parameter with all possible occurring values (one state per value) and all possible transitions. All other FSM (requirement, atomic requirement and system) consist of combinations of Atomic FSM."*

Therefore, each Atomic FSM is dedicated to exactly one system parameter and always has one active value (one state). Communication of System FSM to 'external interaction layer' is achieved with an 'internal interaction layer'. This is shown in Figure 4.18.

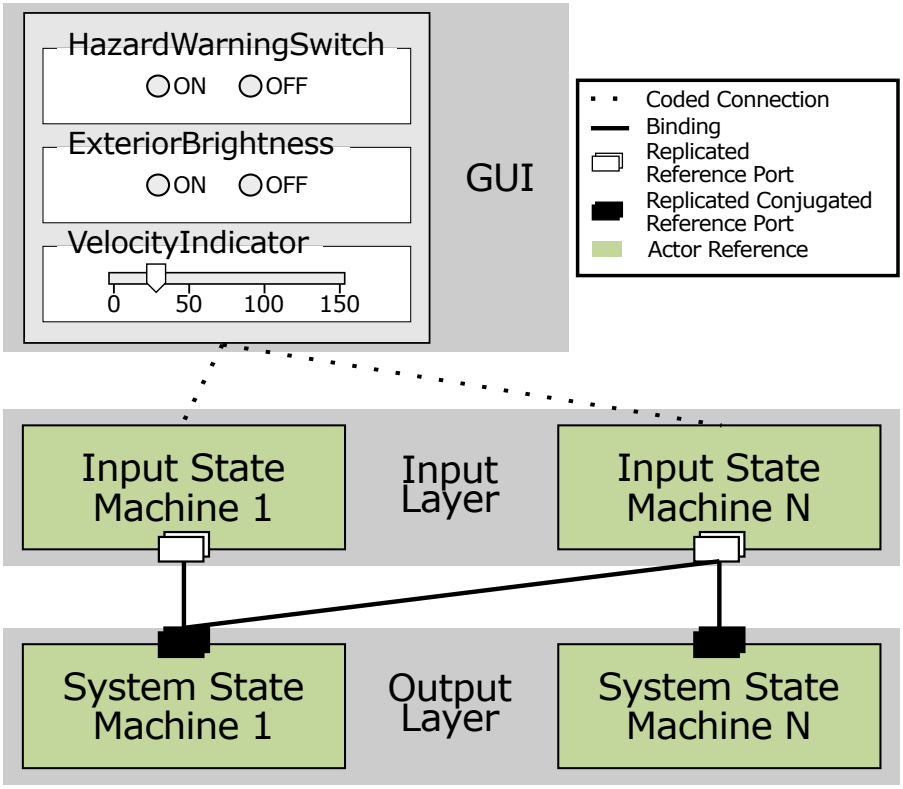


Figure 4.18.:
Dynamic State Machine Generation: Communication Layer, Walter et al. [WMS+19]

This layer receives changes from the input and communicates this change to the 'Atomic FSM', where the input is used and the 'current state' is updated. This new 'current

state' is communicated to the System FSM, where the calculation of the new 'current state' is performed based on input and transition condition. Output is generated based on 'current state' and displayed in the 'external interaction layer'.

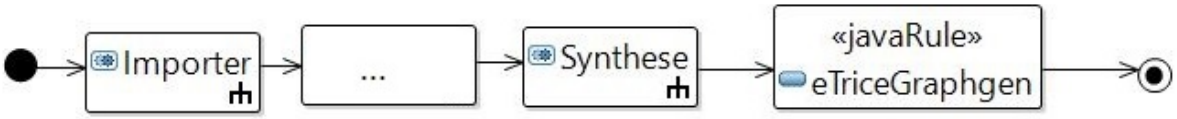


Figure 4.19.:
Dynamic State Machine Generation: Execution Diagram (Extended), (DC43)

The design graph in DC43 is static. Therefore, the execution diagram in DC43 is adjusted with one additional rule: 'eTriceGraphGen'. This is shown in Figure 4.19. The System FSM is generated and the graph generation in 'eTrice' is triggered. This creates the additional execution layer and allows control of the system via the GUI. System interaction can be illustrated with a qualitative example as shown in Walter et al. [WMS+19] and represented in Figure 4.17. Input GUI, Atomic FSM, System FSM and output GUI interact as described before.

Initially, the input is set to *'HazardWarningSwitch[off]'*, therefore Atomic FSM and System FSM have 'current state' equal to *'HazardWarningSwitch[off]'* (System FSM = S_1). All output parameters are set to 'off'. To illustrate a system change, the input is adjusted to *'HazardWarningSwicht[on]'*, which triggers Atomic FSM to *'HazardWarningSwitch[on]'*. Atomic FSM communicates the change to System FSM. The 'current state' input and transition condition are considered and the new 'current state' is changed to S_2. An output is generated. It changes all parameters to *'DirectionIndicator[on]'*. The transition condition is still fulfilled and therefore the new 'current state' is periodically updated. This triggers a change of system output whenever it happens. It can be stopped by providing new input (*'HazardWarningSwitch[off]'*). Generally, such a system control is used to validate the System FSM in the next subsection.

4.4.5. Evaluation

In order to evaluate the created execution layer and to answer **RQ VIII**, the given test cases are executed. Each test case contains test steps where each step has to fulfill a given precondition. Once met, an action is performed. The resulting system reaction is compared to the pass condition.

Table 4.14.: Case Study: Evaluation II (System FSM), Walter et al. [WMS+19]

Function	Passed TS	Blocked TS	Failed TS
Turn and Warning Signaling	36	8	2
Low Beam Headlights	17	0	2
Adaptive (HB) Headlights	4	0	2
Manual (HB) Headlights	4	0	0
Fault Detection	2	0	3
Headlight Technology	0	0	0
Total System	63	8	9

Remark. *Table 4.14 contains the results. Test results are differentiated into three categories. Table 4.12 contains test cases (38) while Table 4.14 contains test steps (80)*

Case (I) - Passed test steps - Passed test steps are successfully executed test steps. These test steps validate a given functionality.

Case (II) - Blocked test steps - Blocked test steps could not be executed and require further discussion. Test steps were blocked due to the missing implementation of the 'bounded existence' pattern and could therefore not be executed. Further discussion is given below.

Case (III) - Failed test steps - Failed test steps did not execute with the expected result but a different system reaction. Therefore, failed test steps require analysis on the impact on the formalization approach. This is given below.

From 80 executed test steps, 63 test steps passed, eight were blocked and nine failed. The blocked and failed test steps were analyzed and a justification for each was found which does not challenge the overall formalization approach. The eight blocked test steps could not be executed. All blocked test steps occurred in the 'Turn and Warning Signal' function. The reason for these test steps being blocked was a missing implementation for the 'bounded existence' pattern (SPS). This pattern is needed to represent the correct functionality for the 'Turn and Warning Signal' in a real-time setup and was not available at time of the validation. The missing implementation was due to limited time available for the code implementation of the conversion algorithm. This reduces the scope of the validation to seven of the eight main patterns but it can be seen as a structural error and is therefore not critical for the overall approach. "The nine failed test steps are distributed among nearly all vehicle functions. A case-wise review and analysis is required. Two failed tests were caused by imprecise structuring of requirements text

at the manual transformation from NL to SPS. The remaining seven test steps can be traced back to incomplete specifications in the initial text form. Blocked tests are caused by the lack of pattern implementation within the approach, yet this does not affect the general validity of the formalization approach. Any incomplete specification prevents the use in this approach. Incomplete specifications cannot be processed. Failed test steps are therefore not caused by the developed formalization process. In addition, as described earlier, blocked tests are caused by the lack of implementation for one specification pattern within the approach, yet this does not affect the general validity." In summary, this shows the general correctness and applicability of the described approach. It allows for an automatic generation of System FSM with an execution layer. The execution layer lets a user control the System FSM and shows the particular effects of a given (added) requirement on the System FSM and its functionality.

5. Conclusion and Outlook

The automotive industry faces a tremendous increase in system complexity. An exponentially rising number of product variants and an increase in dependencies between existing systems (or subsystems) puts pressure on the current state-of-the-art development process. While natural language requirements and test representations must remain, it is obvious, that the given complexity cannot be handled with such a textual representation alone. This work intended to provide a T2M conversion from natural language representation to state machines for automotive requirements and test representations. The goal was set as: "To provide a requirements and test formalization methodology". The consolidation of the achieved results is given in Section 5.1. It summarizes the given work and combines the different outputs towards an overall result. This addresses the initial goal of deriving such a methodology. The limitations to the approach and given results are addressed and discussed in Section 5.2. In Section 5.3, a brief outlook is given. Four different areas of interest are mentioned as potential fields for further research and investigations.

5.1. Conclusion

The central topic of this work was the field of knowledge representations and the transformations between different forms. While the approach was in general a stand-alone methodology, its impact is bigger when paired with other system engineering approaches like digital product design or virtual integration. Therefore, this approach was embedded in an existing graph-based design methodology. It was methodically integrated and used graph-based design as the foundation for its code implementation in the case studies.

The work consolidated the state of the art for the field of graph-based design, automotive requirements and testing as well as selected forms of knowledge representations in Chapter 2. From these existing approaches, a novel requirements and test formalization process chain was derived in Chapter 3. The core idea was presented in Figure 5.1.

An NL expression was converted manually into a specification pattern in SPS. This representation allowed a fully automated further processing. SPS were mapped onto LTL. It was shown how the given data structure can be used to convert LTL to state-wise FOL. The conversion to a structured form like a conjunctive normal form allowed a formalized and machine-readable representation.

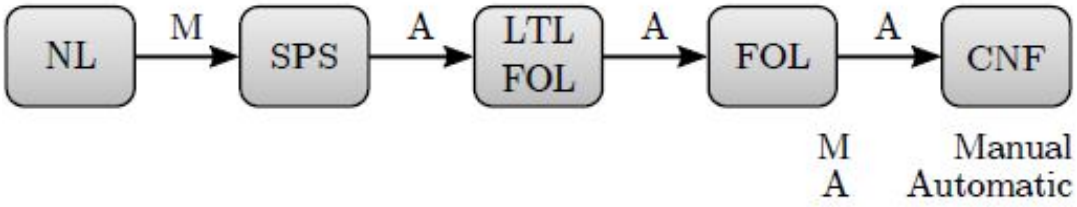


Figure 5.1.: Formalization Process Chain: NL to CNF, Walter et al. [WHPR17]

This process chain was the basis for all derived results in Chapter 4. It was applied in two fields (testing and requirements engineering). Testing data has a simple data structure (here: 'one branch directed graph') while requirements data is more complex (here: 'directed cyclic graph'). Section 4.1 and Section 4.2 use the publications Walter et al. [WHPR17] and Walter et al.[WSPR18] as their foundation while Section 4.3 and Section 4.4 are based on Walter et al. [WMR18] and Walter et al. [WMS+19]. Every section contained and discussed two research questions. The research questions and their answers can be summarized into three general statements: First, it was shown that a step-by-step processing method was derived for testing and requirements engineering. The formalization for testing ends with CNF representation while requirements can be represented in dynamically executable FSM. Second, both approaches were proven to be correct. This was always performed in two separate ways: Each transformation (with the exception of NL to SPS) was shown to be analytically correct. In addition, each overall transformation was empirically tested for correctness and meaningfulness. Third, within in both fields (testing and requirements engineering) practical applications exist, where the presented process adds additional value: In testing, redundant tests were detected, removed and the remaining tests were restructured into an improved test set. The biggest advantage for requirements engineering can be seen in the real-time creation of system state machines based on the given requirements. These derived state machines are controllable and show the exact impact of a requirement onto the system design. This improves the requirements elicitation, documentation and analysis (e.g. through simulations). The approach was shown to be correct (mathematical equivalence

of transformations), applicable (applied to industrial systems) and useful (e.g. reduction of tests in a set, automatic generation of state machines from textual requirements). Certain limitations remain and shall be discussed in the next section.

5.2. Limitations

This section provides an overview of the existing limitations and whether these constrain the approach or not. The most relevant aspect is the expressiveness of all representation forms. SPS can only express relations and actions that can be expressed within the given patterns. This includes all functional requirements and certain NFR (e.g. constraints). Tests are usually in a simpler representation form and no particular problems were observed with processing such tests. LTL constrains the approach in the same manner. Whenever logic classes are required to represent a relation, this approach is not sufficient. There exist further 'SPS to logic' mappings for other logic classes, yet it was not investigated how a logic post-processing for such classes would be performed. Representation in FSM falls into the same category: While many systems can be represented in FSM, certain aspects (e.g. a subset of the overall existing NFR-like requirements addressing the process, quality and regulations) cannot be expressed. No further forms of representation-like activity diagrams and uses-case diagrams were investigated.

In this work, the conversion from LTL to FSM is shown in case-based mappings for particular patterns. A general solution is beyond the scope of this work. However, existing literature like Kam et al. [KVBSV12], Villa et al. [VKBSV13] and Lu and Luo [LL12] discuss general mappings and show that the problem in generality is solvable. Similar to the case-based solution shown for mapping, the aggregation of (requirement) state machines to (system) state machines is shown with a case-based solution through a set of rules. This set is selective and certainly not optimal nor complete. It accomplishes the task required but does not provide a generalized solution. The same sources as above (Kam et al. [KVBSV12], Villa et al. [VKBSV13] and Lu and Luo [LL12]) include generalized solutions for this problem. The case-based approach implemented at the moment does not limit the approach in any noticeable way but might not provide optimal or minimal state machines. The cited works show that in general a solution for the problems of conversion (logic to state machine) and aggregation of the state machines (to system state machines) must exist. The solutions exist in theory but were not specifically derived for this work due to time constrains. A principal (non-optimal) solution

through a rule set was used for the proof of concept. The generalized solutions therefore remain an open research and implementation topic. One concern discussed in Walter et al. [WHPR17] is error introduction. Whenever natural language representations are manually converted into specification patterns, it is possible that the manually given input or the conversion is faulty. Walter et al. [WHPR17] did not observe any occurrences of error introduction in the first case study. Results were crosschecked with the initial requirements from system experts and no errors were found. This does not guarantee the absence of introduced errors since manual work is always prone to error and this work is no exception. Therefore, an almost automated process rather reduces this problem. In case errors are actually (unnoticed) introduced into specification patterns data, these become visible once the System FSM is created and the functionality is observed. Any deviation of observed and expected behavior should trigger a re-investigation of the input data and manual conversion. Therefore, this does not affect the overall approach. Furthermore, the traceability of the requirement specifications helps to identify such faulty conversions enormously.

Error introduction must be addressed in addition for the pre-existing and defined mappings. Temporal logic to simple logic (LTL to FOL) or logic to state machine (FOL to FSM) mappings are analytically provable and therefore the correctness is generally assumed. For errors, it can be shown objectively whether or not an incorrect mapping was provided. The more critical case is the SPS to LTL mapping. This is purely empirical-based and therefore not analytically provable. Dwyer et al. [DAC98, DAC99] derived the mapping and performed extensive empirical studies to validate these mappings. While it is possible that SPS contains incorrect mappings, in the broader sense SPS seems well researched and investigated. Therefore, this does not concern or affect this work in regards to limitations.

The initial data set in NL is readable for non-technical participants. Therefore, a concern is, whether the created expressions and state machines retain a certain readability. Specification patterns are slightly reduced in expressiveness but follow a given structure and therefore do not noticeably degrade in regards to readability. SPS maintains a certain readability while the logic, for example conjunctive normal form is much harder to read. The overall structure must be considered and natural language is replaced with parameters and logic operators. Therefore, readability at least for non-technical people is reduced. Similarly, state machines might express functionality and dependencies better

than plain text, yet this is not readable like unrestricted text. The solution given to this problem is, that the forward conversion can be performed in real-time. All data can be stored in SPS and converted into the formal model with the shown T2M conversion when needed for analysis. In this way, readability for non-technical participants remains while machine-based analysis capability is given. Another concern in regard to readability is scaled systems. This particularly applies for state machines with states and transitions that contain many parameters. For such systems, readability certainly is limited, yet it can be used for machine-based analysis. Classic approaches do not contain an explicit representation of all parameters that affect a given state. The new approach therefore only creates a visibility that was not available prior. Since the representations hold information much more condensed fashion, they were not actually designed for readability. For certain analyses, it is also possible to reduce a state machine to relevant states and relevant parameters. This yields an incomplete system picture but provides for certain readability.

All systems investigated in the case studies were MBC systems. They were all from the 'light' domain. While it was discussed why these systems were chosen and that other domains can be expressed as well, at least the empirical validation is limited to this domain. This is not seen as critical since all conversion steps were performed analytically. It does not affect the overall correctness but might limit the conclusions drawn in regards to the semantic meaning. This was considered and discussed above. Another relevant point to discuss is sample validity. Due to manual processing, only a selected number of tests and requirements was formalized for each system. The first question is, whether this reduced set provides general validity. This was discussed in Walter et al. [WMR18]. The validation is seen as a proof of concept. How well this approach can be used for a given system depends on its general representation capability rather than the selected sample. Second is the concern that bigger systems might not be able to be represented. Scaled systems are certainly even better suited for a formalization since all analyses can be performed mostly in a machine-based rather than a manual manner. A limitation in the reorganization of test cases is, that it must be assumed that no initial sequence contains a given underlying meaning. This assumption is not explicitly expressed. Such meaning would be lost in case of restructuring. It is possible to treat such meanings (if known) by substituting the test steps (with that meaning) into one 'macro' test step. Restructuring is performed and the 'macro' test step is re-substituted to the original multiple test steps.

In summary, the approach can be applied within the expression limits of SPS, LTL and FSM. Mapping from temporal logic to state machines and state machine aggregations are shown case-based for proof of concept. Overall solutions exist but are not expressed explicitly. Error introduction is possible (as in any other manual task), yet not observed in any case study and therefore not perceived as problematic. Readability is reduced for representations in logic and state machine compared to initial representation in natural language. The T2M conversion allows readability in SPS with a later conversion to formal representation with analysis capabilities. Sample validity was discussed due to the selected and reduced systems in the cases studies. Because the analytical proof has been given, the empirical validation is useful for semantic meaningfulness but not necessary for general correctness. Given limitations of the data set are therefore not critical. Overall, the approach is applicable and can be continued in various directions. This is addressed in the next section.

5.3. Outlook

The shown approach represents a theoretical model with applicable industrial use cases. Further investigations can be taken from here in various directions. One relevant question in context of requirements engineering is, how freely requirements can be chosen. Many requirements are given based on physical constraints, while other requirements are specified to achieve a technical solution. The first kind is forced onto a system by necessity while the second kind might be chosen between different technical options. This is discussed in Subsection 5.3.1. Requirements engineering, development and testing are the three big fields in system design. While requirements engineering and testing are addressed in this approach, development is not included. A system design approach with one consistent central model provides many advantages. Subsection 5.3.2 combines the given approach with graph-based design and explores the possibilities of such a unifying approach. The major focus of this work was on deriving a formalized representation. A secondary goal was to perform model optimizations to improve the industrial systems used in the case studies. The effort to create the model was relatively high while automated optimizations are relatively inexpensive. Therefore, further analysis and uses cases for optimization could be investigated. This is briefly addressed in Subsection 5.3.3. In this approach, the exclusive (temporal) logic classes are FOL and LTL. It is possible to extend work beyond these logic classes and consider, for example, ‘computational tree logic’ (CTL) and ‘graphical interval logic’ (GIL) since these already have described

'SPS to logic' mappings from Dwyer et al [DAC17]. Other classes could be introduced as well. Beside the extension of logic classes, many case-based mappings were used for the shown approach. Generalized mappings exist but so far these were not explicitly expressed for this approach. Because this would generalize this work, it this is elaborated in Subsection 5.3.4. These topics only represent a small selection of the seemingly most interesting areas of research. This list is neither complete nor absolute, and many open topics remain and could be investigated.

5.3.1. Deriving Requirements Directly from Physics

Features and attributes for an envisioned system can be represented in requirements. These requirements constrain the solution space for the system design. Whenever another requirement is added, this solution space is reduced. The remaining degrees of freedom are the design choices available for the system design under the given requirements. It is perceived, that requirements can be chosen freely, and while in certain cases this is true, often it is a misconception. Let us assume the following: An envisioned functionality might allow different design solutions, yet the only possibility to connect the required geometrical parts is through a screw connection. There are physical limitations that this connection can withstand and therefore the solution space is therefore affected. It is possible to break this down further and to lay out the physical equations for this screw connection. Let us now also assume a catalog with all available joining technologies and its physical equations, and consequently its physical limits. This represents the envelope within which all system designs must stay in order to work within a physical world. The proposition of this topic is to investigate whether such physical limitations can be expressed explicitly and therefore allow an upfront solution space limitation. It would be expected, that this solution space reduction exceeds the reduction discussed in the beginning since more information (underlying physical principles) is considered. This would make it possible to rule out certain design choices and decisions up front, based on given requirements derived from physical principles.

5.3.2. Automated System Design and Executable V-Model

In system design, the three major fields of engineering are requirements engineering, development and testing. This approach covers requirements engineering and testing. A unifying approach that connects system design in the development phase with the approach shown in this work, would provide a variety of advantages. Walter et al.

[WKR18] proposed such approach in an extended abstract. This was elaborated on in a white paper by Walter et al. [WKR19]. Graph-based design languages are the backbone for one form of digital system design. Knowledge is encoded in class diagram and execution rules. Formalized requirements serve as input which constrains the solution space for the designed system. System geometry is derived through optimization loops against the current system design and the given constraints. Each engineering task has its own design language in its particular domain. In Walter et al. [WKR18], this is shown with the routing of cable wires. All domain languages are applied sequentially against the current system design. When a stable and consistent system design is achieved, graph-based design can verify system robustness through fault tree analysis. In addition, the envisioned approach can perform a digital verification and validation. It verifies consistency of requirements in a static analysis and validates functional correctness of the system by executing the system state machine and compares the behavior to the given requirements. Graph-based design further provides the capability to apply a FEM for mechanical robustness. Such a unified digital development approach has tremendous potential to shorten development cycles and to avoid many problems which occur in classic system design. One digital model contains all system design decisions encoded in a rule set alongside all formalized requirements and tests. This allows for fast adaptations within the solution space and always generates a consistent system design model. Given this, the combination of formal requirements engineering, graph-based system design and formalized digital verification and validation will certainly be explored further.

5.3.3. Further Analysis of Formalized Test Cases and Requirements

In this work, data represented in natural language was formalized into logic expressions and FSM representations. The initial effort necessary to convert the data is rather high. The data was optimized for redundancies and execution order of test steps. The improvements for the new data set were justified with a comparison of formalization effort and manual optimization effort. Further optimizations would not require formalization effort but would simply need a one-time implementation of the optimization method. This effort is minor compared to the mentioned initial formalization effort. Therefore, it seems practical to make use of the existing formal representation and include further applications. Redundancy is already included and consistency is implicitly accounted for. Whenever a state in a state machine has inconsistent parameters (e.g. Light[on] AND Light[off]), an inconsistency is given. So far, no machine-based investigation is

undertaken to identify whether or not requirements are inconsistent. Another use case would be to show the minimal set of requirements: removing redundant and inconsistent requirements would provide such a set. A more challenging task would be to show the completeness of requirements. Created state machines and envisioned functionality must be compared by execution of the dynamic state machine and observation of the system behavior. This might not be fully automatable, yet such approaches could show completeness for given requirements. To show completeness for tests, on the other hand, an automated approach is certainly possible. Existing tests must be linked to the states and transitions (inheritance of dependencies through requirements). It must be checked whether a given change criterion (state, branch or mc/dc coverage) is given and aimed for. Missing tests could be pointed out (and could potentially be created). As shown with this selection of applications, many useful data optimizations are possible and could be included with rather small effort. This would certainly benefit the industrial applications.

5.3.4. Extension of Case-Based LTL to FOL Mapping

The major limitation to the formalization process is expansion of each representation form. Linear temporal logic can express temporal relations on one 'directed time branch'. While this covers the majority of uses cases for the automotive system under investigation, other automotive domains and other industries might require further expressiveness. Two of many candidates for further logic classes are Computational Tree Logic (CTL) and Graphical Interval Logic (GIL). These are selected for this brief discussion due to the fact that 'SPS to logic' mapping for CTL and GIL already exists within Dwyer et al. [DAC17]. Computational tree logic is capable of representing branching time logic. It contains an uncertainty about a future event and therefore has a tree structure for branching. Graphical interval logic can express sequential and parallel time relations. Both extend the given approach in regards to overall expressiveness. The effects on state machine representation must be analyzed and solved in order to include these in the approach. Another extension for the given approach is the generalization of the 'logic to FSM' mapping. The approach uses case-based mappings which are provided for all SPS paths in the Appendix B. There exist generalized solutions, but this work only used the given case-based mappings due to scope and time constraints. A generalized solution provides a significant advantage. Any specification path that can be mapped to logic can be used in the approach. This would extend the field of applications significantly. Approaches that prefer any other pattern over SPS could define 'pattern to logic' map-

ping and apply all further steps of this approach. The optimization of the state machines remains open as well. Aggregations of requirement state machines to a system state machine were performed with a small set of explicit rules ('atomization' and 'minimization' (including 'merge states', 'merge transitions' and 'add links')). This is a simple approach that worked for the given problem set. This rule set is certainly not optimal. A generalization exists, as described in abstract form in literature like Kam et al. [KVBSV12], Villa et al. [VKBSV13] or Lu and Luo [LL12]. These abstract solutions need to be adjusted to this particular problem. Such an adaptation would allow the derivation of optimal (minimal) state machines, which would simplify and improve representation and analysis. These four topics represent a selection of many open fields. These were chosen for the following reason: Subsection 5.3.1 is the authors choice for further scientific research. Subsection 5.3.2 connects this work with the graph-based design and particularly with the work by Rudolph. The extension in Subsection 5.3.3 is the path to increase the industrial impact on the particular projects where the case studies were performed. The topic of Subsection 5.3.4 describes for the most part tasks that would strengthen the approach without extending the scope but generalizing the given modeling. While these represent a few out of many fields for potential further research, this section is the closing section of this work. Therefore, it shall be used to summarize and close this work:

The overall premise of this work was to derive a formalization method for automotive requirements and testing data. This was achieved and laid out in the previous chapters. While many open questions remain, a small contribution to knowledge increase was made with this work and therefore this work shall close with a quote:

"In a dark place we find ourselves, and a little more knowledge lights our way." Yoda

A. Mapping SPS to LTL (full)

The appendix contains three parts. Appendix A describes all mappings made in Dwyer et al. [DAC98, DAC99] for SPS to LTL. Appendix B provides case-based mappings for all LTL expressions that arise from the SPS. For each LTL expression, its representation in the form of a FSM is shown. Lastly, in Appendix E, the authors publications are mentioned and aligned.

This chapter contains the full 'SPS to LTL' mapping as developed and validated by Dwyer et al. [DAC98, DAC99, DAC17]. It represents the extension of Subsection 3.3.2 where a selected set of mappings is shown. Overall, there exist eleven patterns with five scopes each.

Remark. *SPS representation for each entry is created by adding the pattern (f.e. 'P is true') with the scope (f.e. 'between Q and R'). This creates the SPS representation 'P is true between Q and R'.*

Universality - P is true

Table A.1.: Mapping: SPS to LTL (Pattern: 'Universality')

Scope	Linear Temporal Logic
Globally	$\Box\ P$
Before R	$\diamond\ R \longrightarrow (P\ U\ R)$
After Q	$\Box\ (Q \longrightarrow \Box\ (P))$
Between Q and R	$\Box\ ((Q \wedge \diamond\ R) \longrightarrow P\ U\ R)$
After Q until R	$\Box\ (Q \longrightarrow P\ U\ (R \vee \Box\ (P)))$

Absence - P is false

Table A.2.: Mapping: SPS to LTL (Pattern: 'Absence')

Scope	Linear Temporal Logic
Globally	$\Box \neg P$
Before R	$\diamond R \longrightarrow (\neg P \; U \; R)$
After Q	$\Box (Q \longrightarrow \Box (\neg P))$
Between Q and R	$\Box ((Q \wedge \diamond R) \longrightarrow \neg P \; U \; R)$
After Q until R	$\Box (Q \longrightarrow \neg P \; U \; (R \vee \Box (\neg P)))$

Existence - P becomes true

Table A.3.: Mapping: SPS to LTL (Pattern: 'Existence')

Scope	Linear Temporal Logic
Globally	$\diamond P$
Before R	$\diamond R \longrightarrow (R \; U \; P)$
After Q	$\Box (\neg Q \vee \diamond (Q \wedge \diamond P))$
Between Q and R	$\Box ((Q \wedge \diamond R) \longrightarrow R \; U \; P)$
After Q until R	$\Box (Q \longrightarrow (\neg R \; U \; P))$

Bounded Existence - P becomes true
(Transitions to P-state occur at most 2 times)

Table A.4.: Mapping: SPS to LTL (Pattern: 'Bounded Existence')

Scope	Linear Temporal Logic
Globally	$(\neg PW(PW(\neg PW(PW\Box\neg P)))$ where $PWQ = \Box P \vee (PUQ)$
Before R	$\diamond R \longrightarrow ((\neg P \wedge \neg R)U(R \vee ((P \wedge \neg R)U(R \vee ((\neg P \wedge \neg R) U(R \vee ((P \wedge \neg R)U(R \vee (\neg PUR)))))$
After Q	$\diamond Q \longrightarrow (\neg QU(Q \wedge (\neg PW(PW(\neg PW(PW\Box\neg P)))))$
Between Q and R	$\Box((Q \wedge \diamond R) \longrightarrow ((\neg P \wedge \neg R)U(R \vee ((P \wedge \neg R)U(R \vee ((\neg P \wedge \neg R)U(R \vee ((P \wedge \neg R)U(R \vee (\neg PUR))))))$
After Q until R	$\Box Q \longrightarrow ((\neg P \wedge \neg R)U(R \vee ((P \wedge \neg R)U(R \vee ((\neg P \wedge \neg R) U(R \vee ((P \wedge \neg R)U(R \vee (\neg PW R) \vee \Box P))))$

Response - S responds to P

Table A.5.: Mapping: SPS to LTL (Pattern: 'Response')

Scope	Linear Temporal Logic
Globally	$\Box(P \longrightarrow \diamond S)$
Before R	$(P \longrightarrow (\neg RUS))U(R \vee \Box(\neg R))$
After Q	$\Box(Q \longrightarrow \Box(P \longrightarrow \diamond S))$
Between Q and R	$\Box((Q \wedge \diamond R) \longrightarrow (P \longrightarrow (\neg RUS))UR)$
After Q until R	$\Box(Q \longrightarrow ((P \longrightarrow (\neg RUS))UR) \vee \Box(P \longrightarrow (\neg RUS)))$

Response Chain I - P responds to S,T

2 stimulus - 1 response chain

Table A.6.: Mapping: SPS to LTL (Pattern: 'Response Chain I')

Scope	Linear Temporal Logic
Globally	$\Box(P \longrightarrow \diamond S)$
Before R	$\diamond R \longrightarrow (S \wedge \circ(\neg RUT) \longrightarrow \circ(\neg RU(T \wedge \diamond P)))UR$
After Q	$\Box(Q \longrightarrow \Box(S \wedge \circ \diamond T \longrightarrow \circ(\neg TU(T \wedge \diamond P))))$
Between Q and R	$\Box((Q \wedge \diamond R) \longrightarrow (S \wedge \circ(\neg RUT) \longrightarrow \circ(\neg RU(T \wedge \diamond P)))UR$
After Q until R	$\Box(Q \longrightarrow (S \wedge \circ(\neg RUT) \longrightarrow \circ(\neg RU(T \wedge \diamond P)))$ $U(R \vee \Box(S \wedge \circ(\neg RUT) \longrightarrow \circ(\neg RU(T \wedge \diamond P)))))$

Response Chain II - S,T responds to P

1 stimulus - 2 response chain

Table A.7.: Mapping: SPS to LTL (Pattern: 'Response Chain II')

Scope	Linear Temporal Logic
Globally	$\Box(P \longrightarrow \diamond(S \wedge \circ \diamond T))$
Before R	$\diamond R \longrightarrow (P \longrightarrow (\neg RU(S \wedge \neg R \wedge \circ((\neg R \wedge UT))))UR$
After Q	$\Box(Q \longrightarrow (P \longrightarrow (S \wedge \circ \diamond T)))$
Between Q and R	$\Box((Q \wedge \diamond R) \longrightarrow (P \longrightarrow (\neg RU(S \wedge \neg R \wedge \circ((\neg R \wedge UT))))UR)$
After Q until R	$\Box(Q \longrightarrow (P \longrightarrow (\neg RU(S \wedge \neg R \wedge \circ((\neg R \wedge UT))))U$ $(R \vee \Box(P \longrightarrow (S \wedge \circ \diamond T))))$

Precedence S precedes P

Table A.8.: Mapping: SPS to LTL (Pattern: 'Precedence')

Scope	Linear Temporal Logic
Globally	$\diamond P \longrightarrow (\neg P U (S \wedge \neg P))$
Before R	$\diamond P \longrightarrow (\neg P U (S \vee R))$
After Q	$\square \neg Q \vee \diamond (Q \wedge (\neg P U (S \vee \square \neg P)))$
Between Q and R	$\square ((Q \wedge \diamond R) \longrightarrow (\neg P U (S \vee R)))$
After Q until R	$\square Q \longrightarrow ((\neg P U (S \vee R)) \vee \square \neg P))$

Precedence Chain I - S,T precedes P
2 causes - 1 effect precedence chain

Table A.9.: Mapping: SPS to LTL (Pattern: 'Precedence Chain I')

Scope	Linear Temporal Logic
Globally	$\diamond P \longrightarrow (\neg P U (S \wedge \neg P \wedge \circ(\neg P U T)))$
Before R	$\diamond R \longrightarrow (\neg P U (R \vee (S \wedge \neg P \wedge \circ(\neg P U T))))$
After Q	$\square \neg Q \vee (\neg Q U (Q \wedge \diamond P \longrightarrow (\neg P U (S \wedge \neg P \wedge \circ(\neg P U T)))))$
Between Q and R	$\square ((Q \wedge \diamond R) \longrightarrow (\neg P U (R \vee (S \wedge \circ(\neg P U T)))))$
After Q until R	$\square (Q \longrightarrow (\diamond P \longrightarrow (\neg P U (R \vee (S \wedge \circ(\neg P U T))))))$

Precedence Chain II - P precedes (S,T)
1 cause - 2 effects precedence chain

Table A.10.: Mapping: SPS to LTL (Pattern: 'Precedence Chain II')

Scope	Linear Temporal Logic
Globally	$(\diamond(S \wedge \circ \diamond T)) \longrightarrow ((\neg S) U P)$
Before R	$\diamond R \longrightarrow ((\neg(S \wedge (\neg R)) \wedge \circ(\neg R U (T \wedge \neg R))))U(R \vee P))$
After Q	$(\square \neg Q) \vee ((\neg Q) U (Q \wedge ((\diamond(S \wedge \circ \diamond T)) \longrightarrow ((\neg S) U P))$
Between Q and R	$\square ((Q \wedge \diamond R) \longrightarrow ((\neg(S \wedge (\neg R)) \wedge \circ(\neg R U (T \wedge \neg R))))U(R \vee P))$
After Q until R	$\square (Q \longrightarrow (\neg(S \wedge (\neg R) \wedge \circ(\neg R(T \wedge \neg R)))U (R \vee P) \vee \square(\neg(S \wedge \circ \diamond T))))$

Constrained Chain Patterns - S,T without Z responds to P

Table A.11.: Mapping: SPS to LTL (Pattern: 'Constrained Chain Pattern')

Scope	Linear Temporal Logic
Globally	$\Box(P \longrightarrow \diamond(S \wedge \neg Z \wedge \circ(\neg ZUT)))$
Before R	$\diamond R \longrightarrow (P \longrightarrow (\neg RU(S \wedge \neg R \wedge \neg Z \wedge$ $\circ((\neg R \wedge \neg Z)UT))))UR$
After Q	$\Box(Q \longrightarrow \Box(P \longrightarrow (S \wedge \neg Z \wedge \circ(\neg ZUT))))$
Between Q and R	$\Box((Q \wedge \diamond R) \longrightarrow (P \longrightarrow (\neg RU(S \wedge \neg R \wedge \neg Z \wedge$ $\circ((\neg R \wedge \neg Z)UT))))UR)$
After Q until R	$\Box(Q \longrightarrow (P \longrightarrow (\neg RU(S \wedge \neg R \wedge \neg Z \wedge \circ((\neg R \wedge \neg Z)UT))))$ $U(R \vee \Box(P \longrightarrow (S \wedge \neg Z \wedge \circ(\neg ZUT)))))$

Remark: The different representation of logic operators in this work versus Dwyer's publications [DAC17, DAC98, DAC99] is shown in Table 3.3

B. Mapping LTL to FSM (full)

This chapter contains the full 'LTL to FSM' mapping as developed and validated by Walter et al. [WMR18]. It represents the extension of Subsection 3.5.2 where a selected set of mappings is shown. Overall, there exist mappings for ten patterns with five scopes each (mapping for 'Chain Response' is not included.). The patterns are sorted in terms of its textual description, which stands in a 1:1 relationship to the mapped LTL expressions.

Remark. *: SPS representation for each entry is created by adding the pattern (f.e. 'P is true') with the scope (f.e. 'between Q and R'). This creates the SPS representation 'P is true between Q and R'.*

Pattern: Universality - P is true

Globally

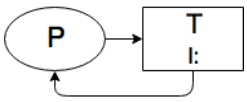


Figure B.1.: Mapping LTL to FOL conversion - Universality 'Global' [WMS+19]

Before R

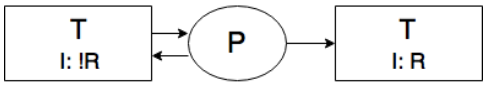


Figure B.2.: Mapping LTL to FOL conversion - Universality 'before R' [WMS+19]

After Q

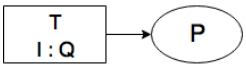


Figure B.3.: Mapping LTL to FOL conversion - Universality 'After Q' [WMS+19]

Between Q and R

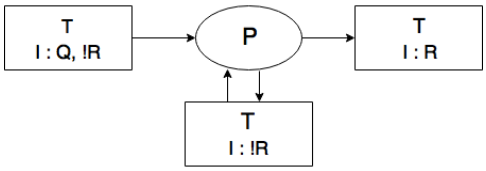


Figure B.4.: Mapping LTL to FOL conversion - Universality 'Between Q and R' [WMS+19]

After Q until R

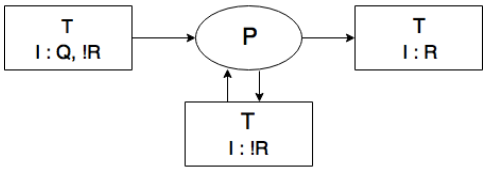


Figure B.5.: Mapping LTL to FOL conversion - Universality 'After Q until R' [WMS+19]

Pattern: Absence - P is false

Globally

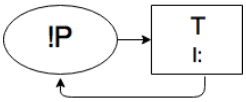


Figure B.6.: Mapping LTL to FOL conversion - Absence 'Global' [WMS+19]

Before R

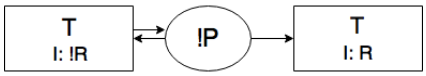


Figure B.7.: Mapping LTL to FOL conversion - Absence 'before R' [WMS+19]

After Q

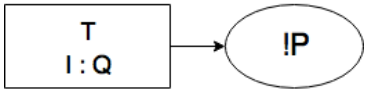


Figure B.8.: Mapping LTL to FOL conversion - Absence 'After Q' [WMS+19]

Between Q **and** R

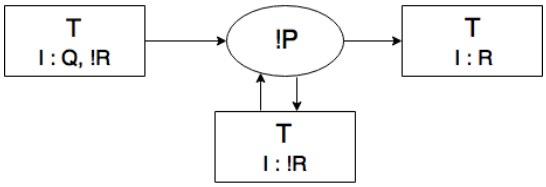


Figure B.9.: Mapping LTL to FOL conversion - Absence 'Between Q and R' [WMS+19]

After Q **until** R

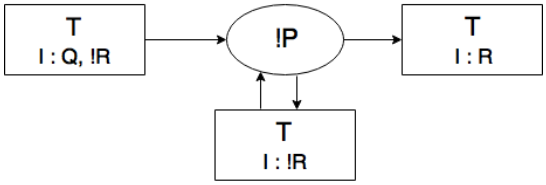


Figure B.10.: Mapping LTL to FOL conversion - Absence 'After Q until R' [WMS+19]

Pattern: Existence - P becomes true

Globally

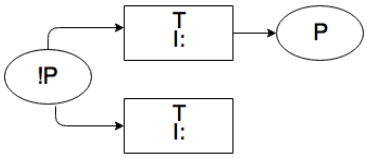


Figure B.11.: Mapping LTL to FOL conversion - Existence 'Global' [WMS+19]

Before R

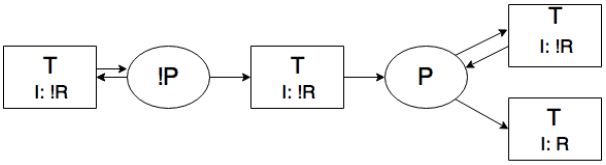


Figure B.12.: Mapping LTL to FOL conversion - Existence 'before R' [WMS+19]

After Q

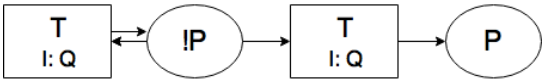


Figure B.13.: Mapping LTL to FOL conversion - Existence 'After Q' [WMS+19]

Between Q and R

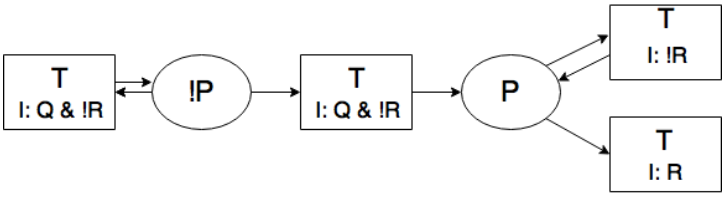


Figure B.14.: Mapping LTL to FOL conversion - Existence 'Between Q and R' [WMS+19]

After Q until R

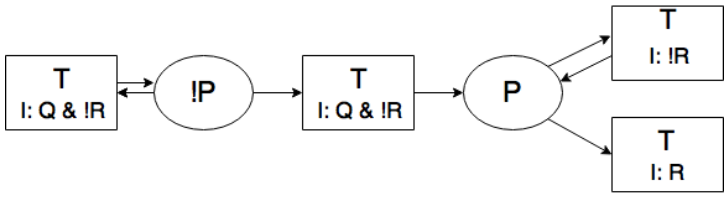


Figure B.15.: Mapping LTL to FOL conversion - Existence 'After Q until R' [WMS+19]

Pattern: Bounded Existence - P becomes true
(Transitions to P-state occur at most 2 times)

Globally

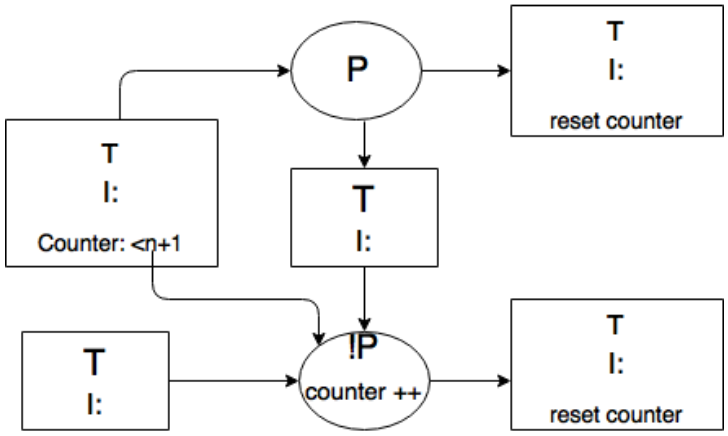


Figure B.16.: Mapping LTL to FOL conversion - Bounded Existence 'Global' [WMS+19]

Before R

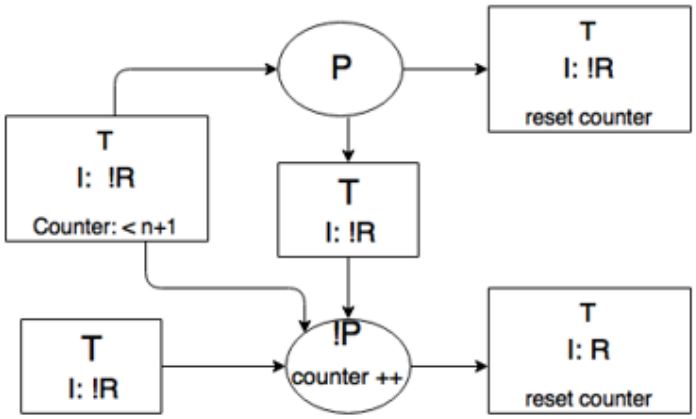


Figure B.17.: Mapping LTL to FOL conversion - Bounded Existence 'before R' [WMS+19]

After Q

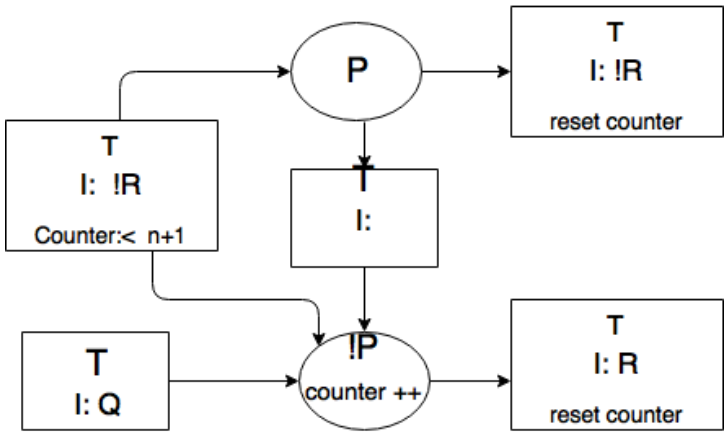


Figure B.18.: Mapping LTL to FOL conversion - Bounded Existence 'After Q' [WMS+19]

Between Q and R

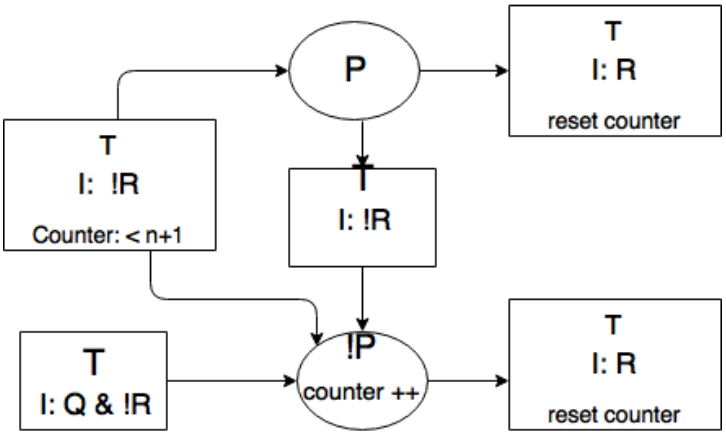


Figure B.19.: Mapping LTL to FOL conversion - Bounded Exist. 'Between Q and R' [WMS+19]

After Q until R

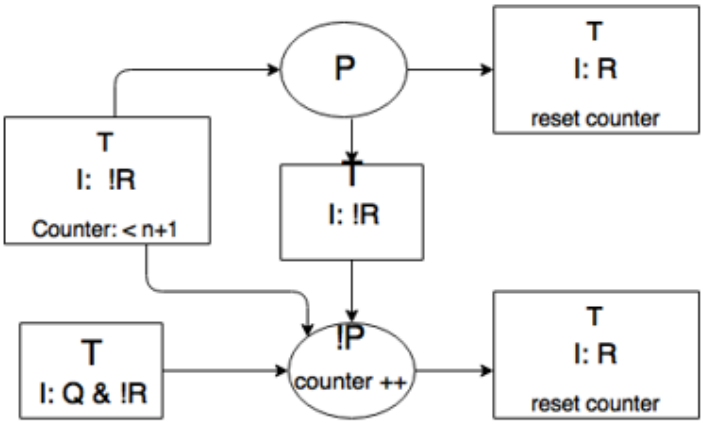


Figure B.20.: Mapping LTL to FOL conversion - Bounded Exist. 'After Q until R' [WMS+19]

Pattern: Response - S responds to P

Globally

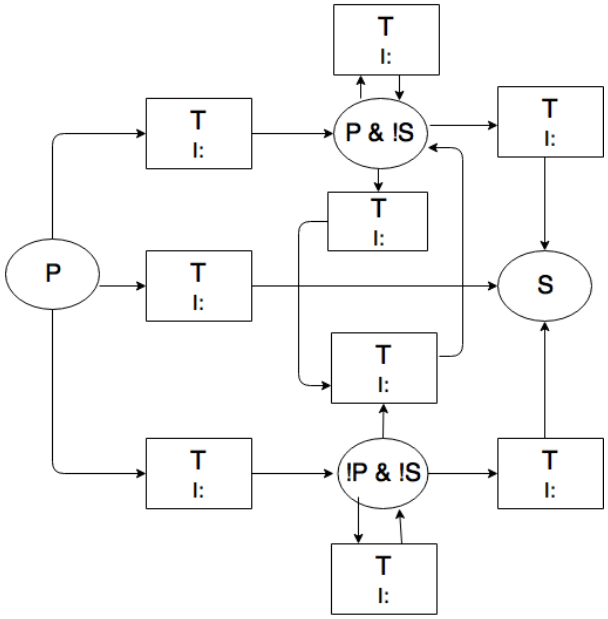


Figure B.21.: Mapping LTL to FOL conversion - Response 'Global' [WMS+19]

Before R

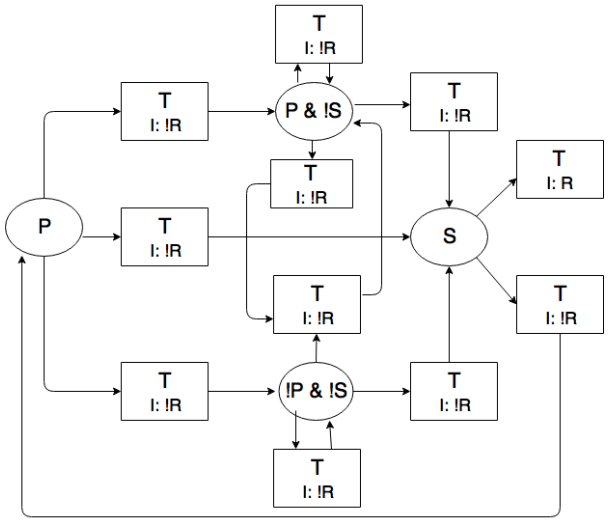


Figure B.22.: Mapping LTL to FOL conversion - Response 'before R' [WMS+19]

After Q

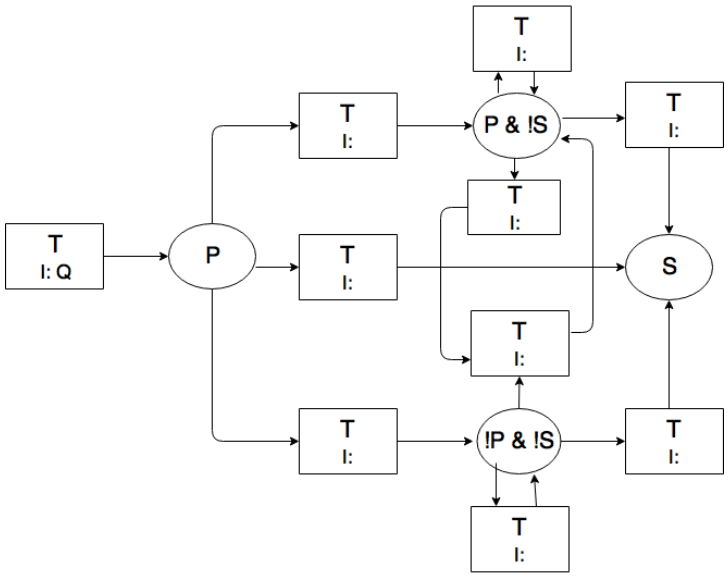


Figure B.23.: Mapping LTL to FOL conversion - Response 'After Q' [WMS+19]

Between Q **and** R

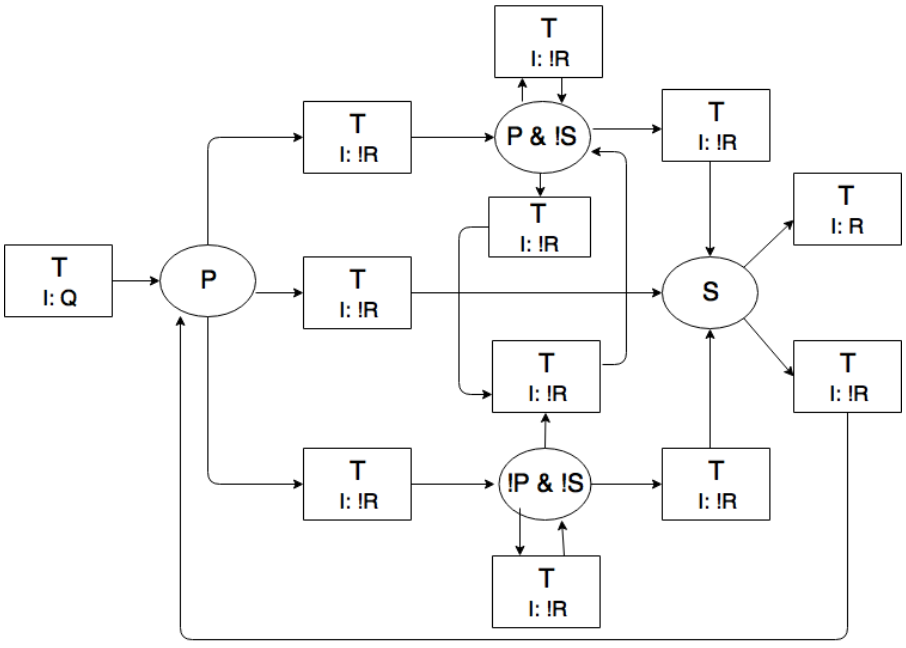


Figure B.24.: Mapping LTL to FOL conversion - Response 'Between Q and R' [WMS+19]

After Q until R

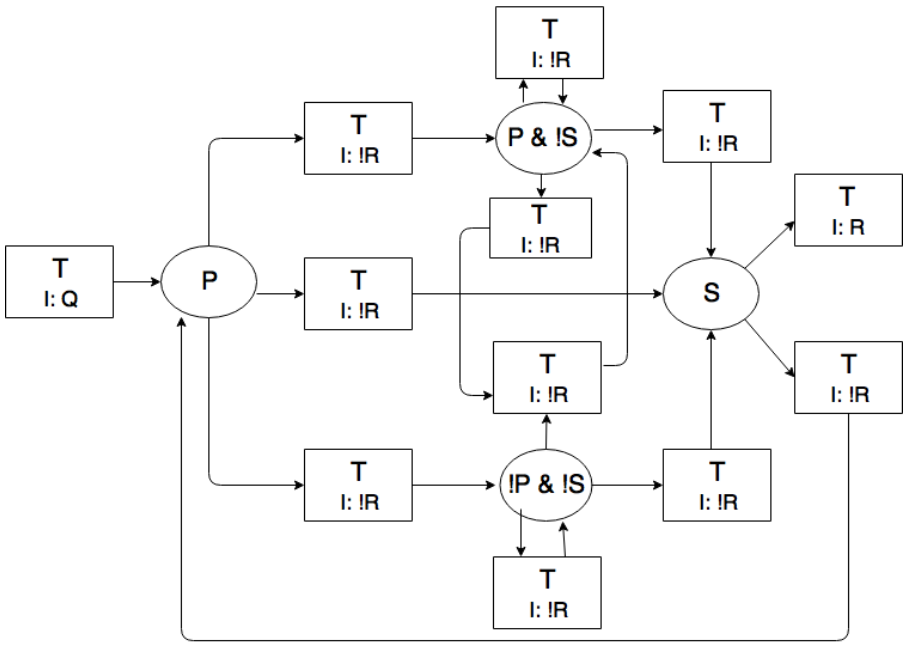


Figure B.25.: Mapping LTL to FOL conversion - Response 'After Q until R' [WMS+19]

Pattern: Response Chain I - S responds to P, T
2 stimulus - 1 response chain

Globally

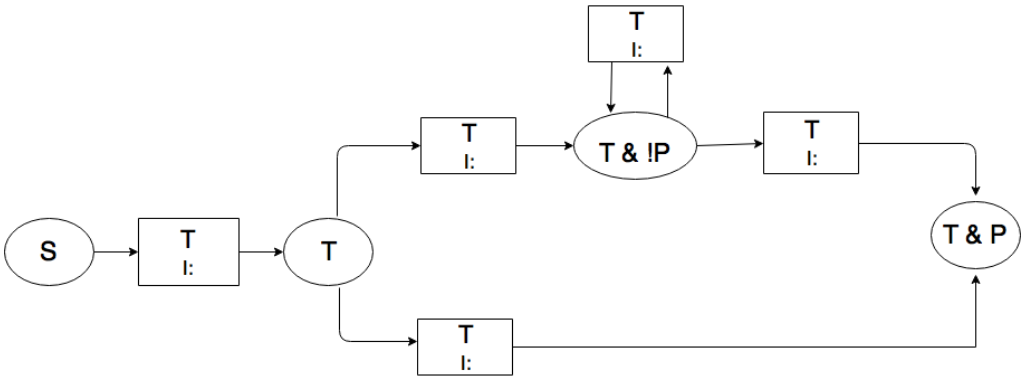


Figure B.26.: Mapping LTL to FOL conversion - Response Chain I 'Global' [WMS+19]

Before R

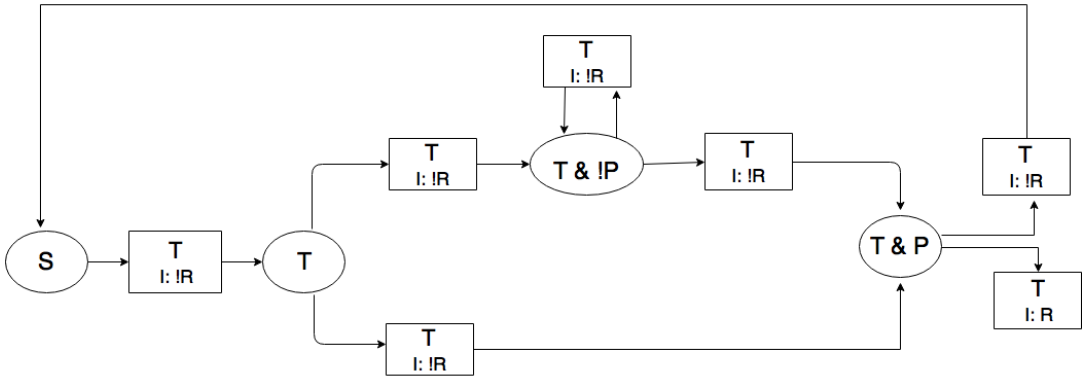


Figure B.27.: Mapping LTL to FOL conversion - Response Chain I 'before R' [WMS+19]

After Q

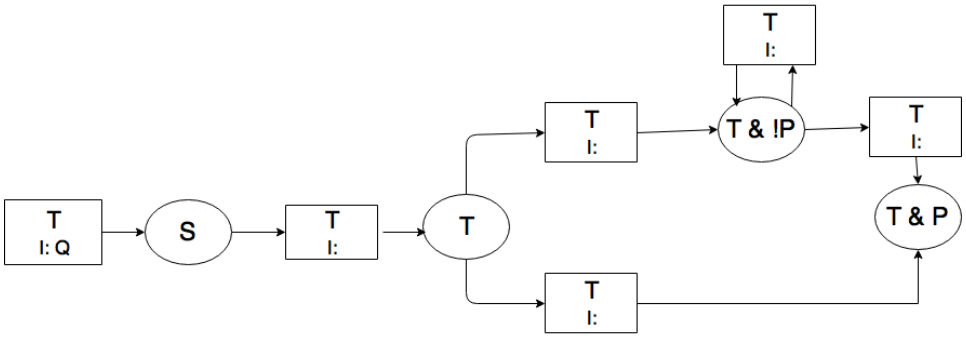


Figure B.28.: Mapping LTL to FOL conversion - Response Chain I 'After Q' [WMS+19]

Between Q and R

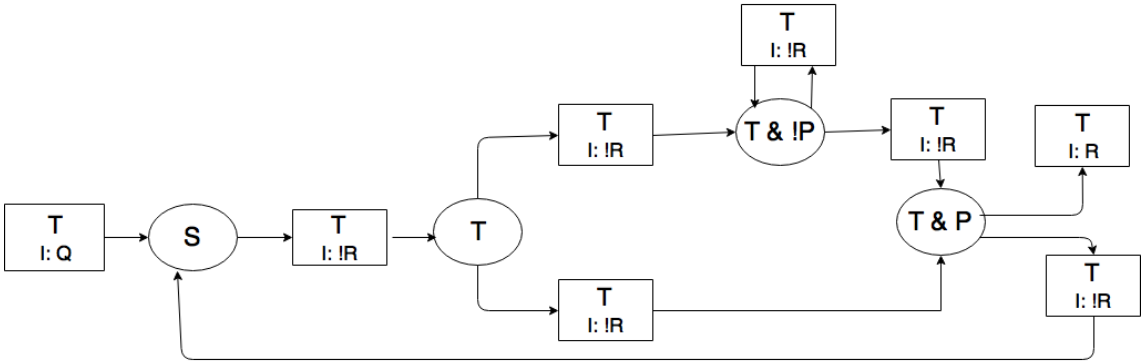


Figure B.29.: Mapping LTL to FOL conversion - Resp. Chain I 'Between Q and R' [WMS+19]

After Q until R

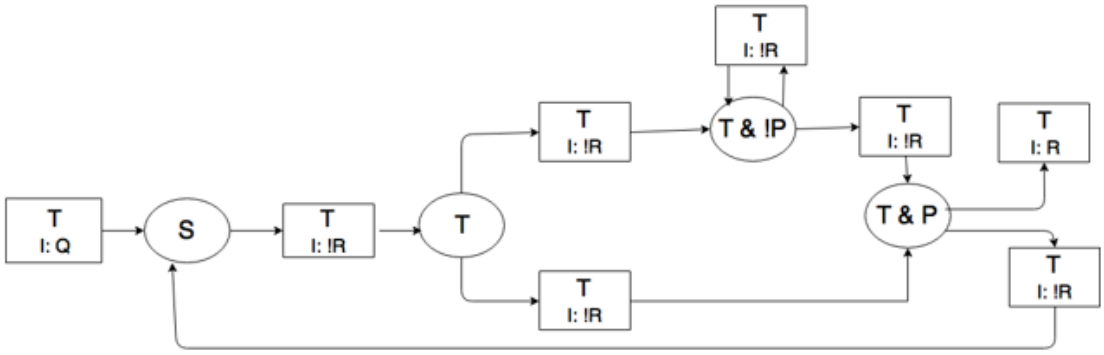


Figure B.30.: Mapping LTL to FOL conversion - Response Chain I 'After Q until R' [WMS+19]

Pattern: Response Chain II - S responds to P, T
1 stimulus - 2 response chain

Globally

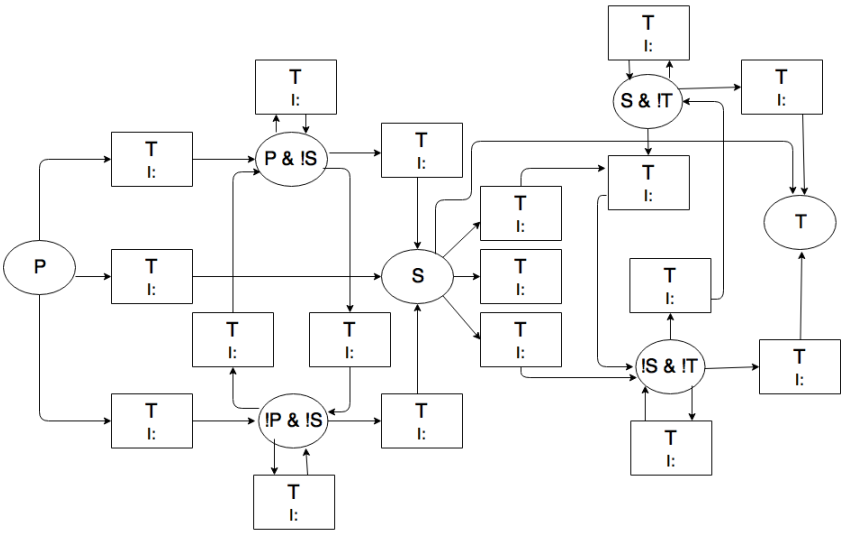


Figure B.31.: Mapping LTL to FOL conversion - Response Chain II 'Global' [WMS+19]

Before R

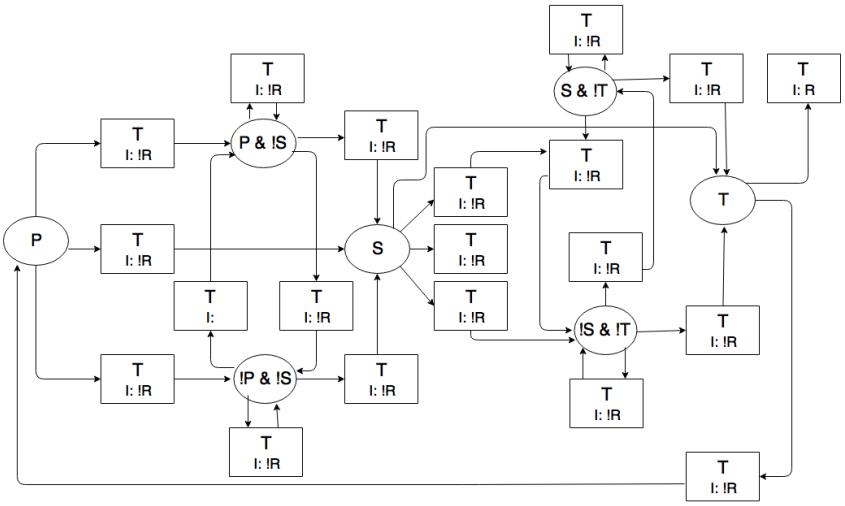


Figure B.32.: Mapping LTL to FOL conversion - Response Chain II 'before R' [WMS+19]

After Q

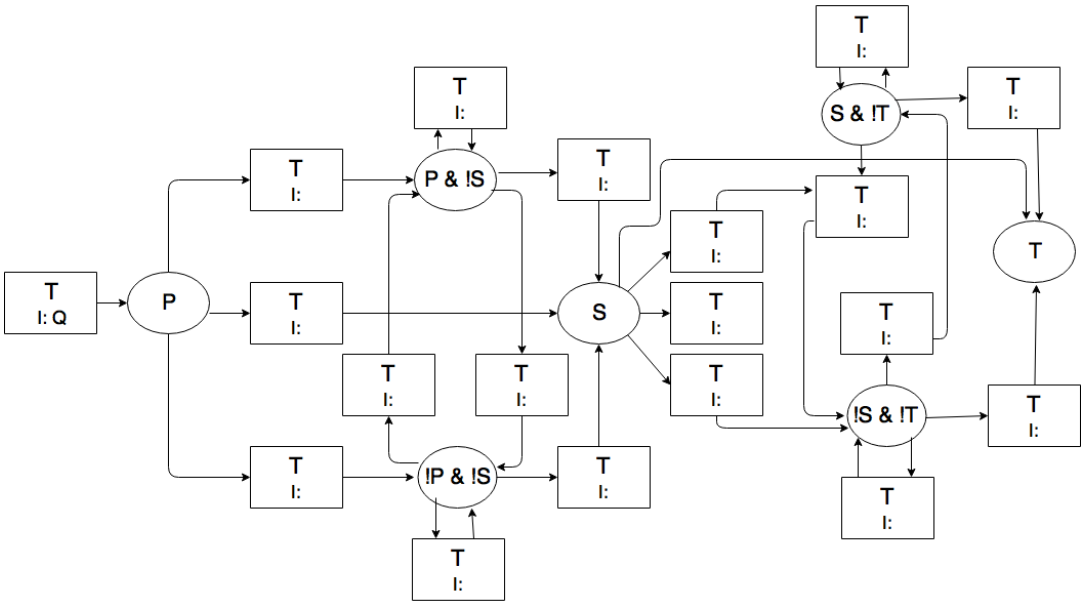


Figure B.33.: Mapping LTL to FOL conversion - Response Chain II 'After Q' [WMS+19]

Between Q and R

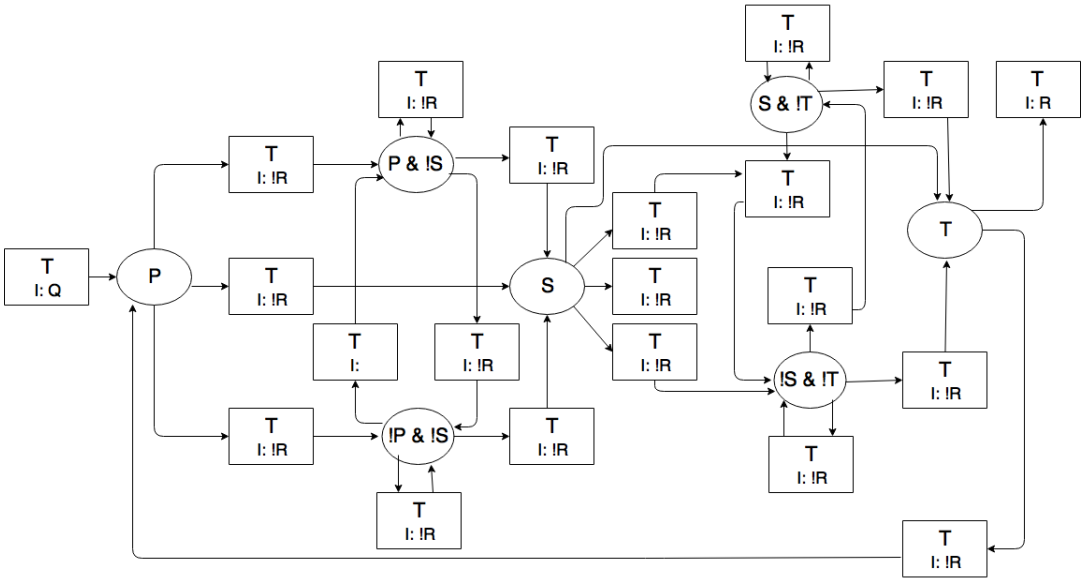


Figure B.34.: Mapping LTL to FOL conversion - Resp. Chain II 'Between Q and R' [WMS+19]

After Q until R

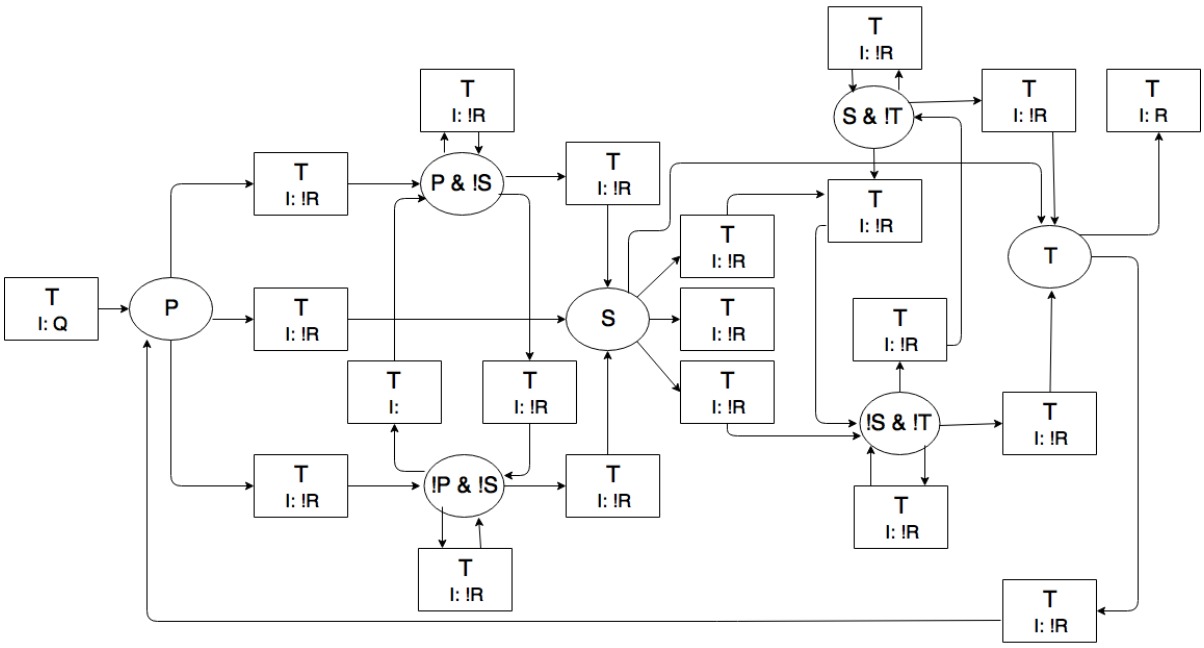


Figure B.35.: Mapping LTL to FOL conversion - Response Chain II 'After Q until R' [WMS+19]

Pattern: Precedence - S precedes P

Globally

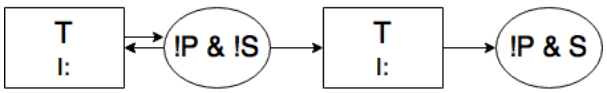


Figure B.36.: Mapping LTL to FOL conversion - Precedence 'Global' [WMS+19]

Before R

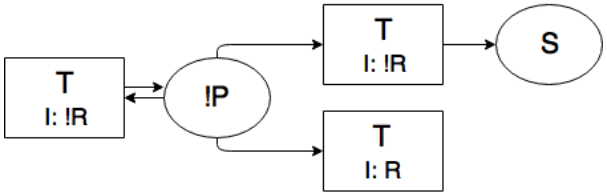


Figure B.37.: Mapping LTL to FOL conversion - Precedence 'before R' [WMS+19]

After Q

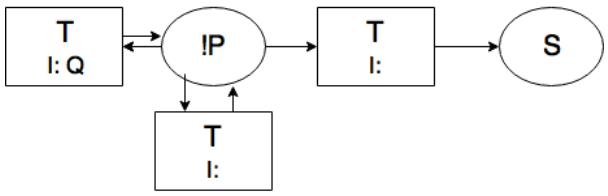


Figure B.38.: Mapping LTL to FOL conversion - Precedence 'After Q' [WMS+19]

Between Q and R

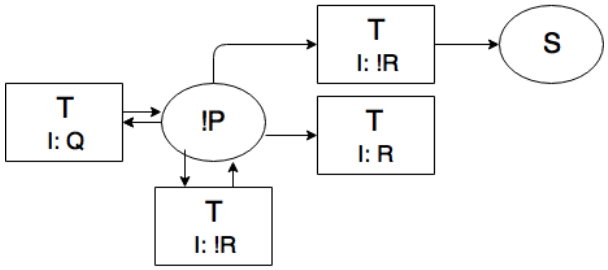


Figure B.39.: Mapping LTL to FOL conversion - Precedence 'Between Q and R' [WMS+19]

After Q until R

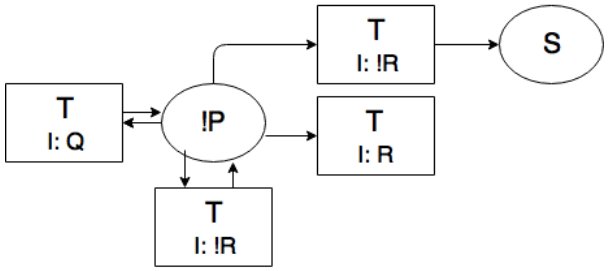


Figure B.40.: Mapping LTL to FOL conversion - Precedence 'After Q until R' [WMS+19]

Pattern: Precedence Chain I - S,T precedes P
2 causes - 1 effect precedence chain

Globally

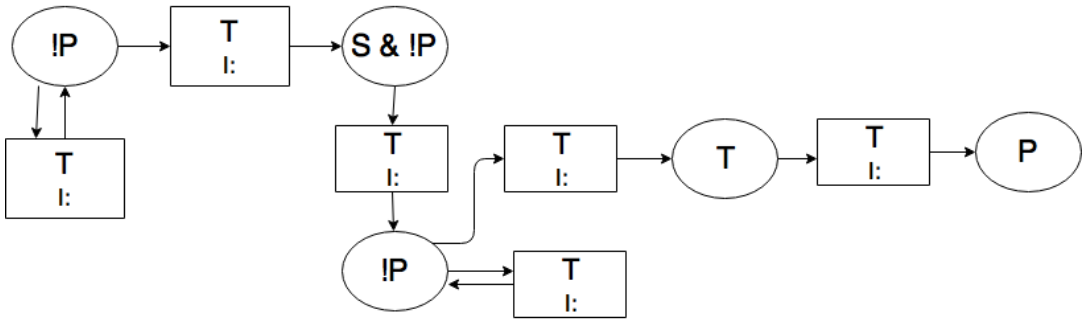


Figure B.41.: Mapping LTL to FOL conversion - Precedence Chain I 'Global' [WMS+19]

Before R

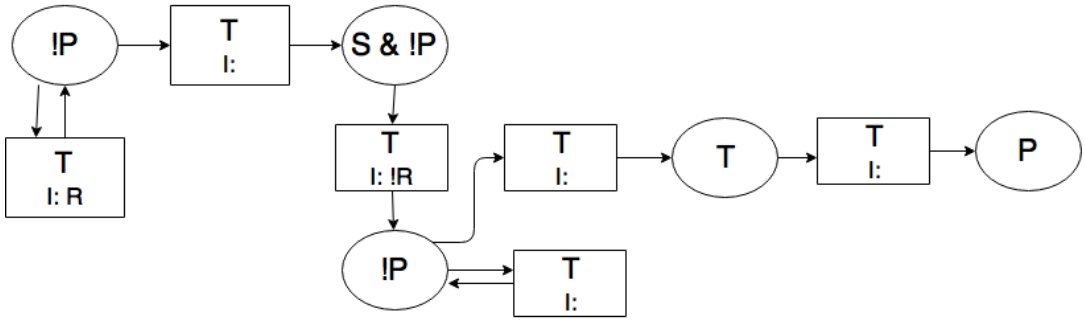


Figure B.42.: Mapping LTL to FOL conversion - Precedence Chain I 'before R' [WMS+19]

After Q

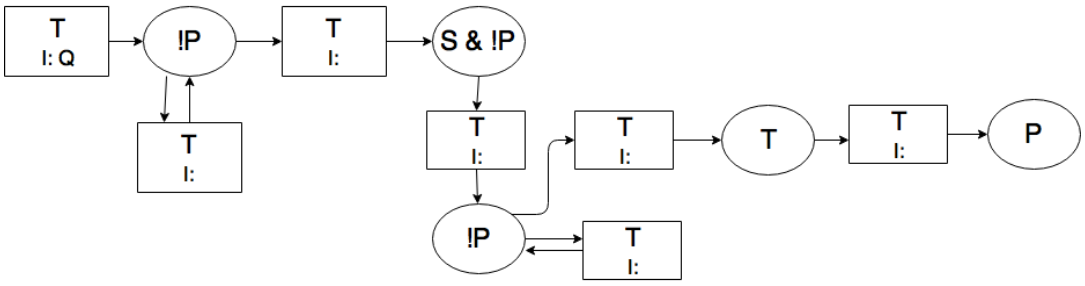


Figure B.43.: Mapping LTL to FOL conversion - Precedence Chain I 'After Q' [WMS+19]

Between Q and R

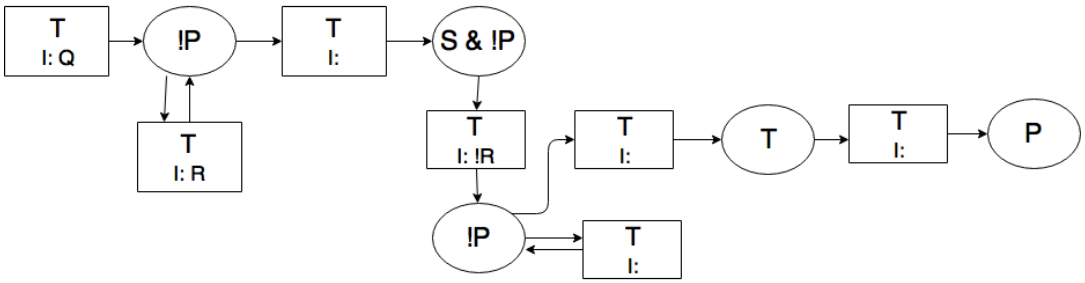


Figure B.44.: Mapping LTL to FOL conversion - Preced. Chain I 'Between Q and R' [WMS+19]

After Q until R

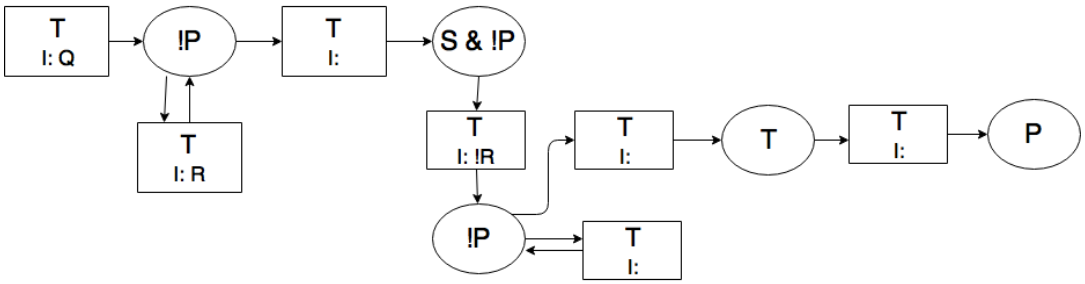


Figure B.45.: Mapping LTL to FOL conversion - Preced. Chain I 'After Q until R' [WMS+19]

Pattern: Precedence Chain II - S,T precedes P
1 causes - 2 effect precedence chain

Globally

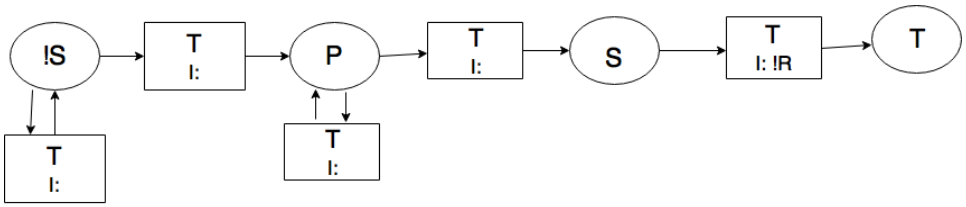


Figure B.46.: Mapping LTL to FOL conversion - Precedence Chain II 'Global' [WMS+19]

Before R

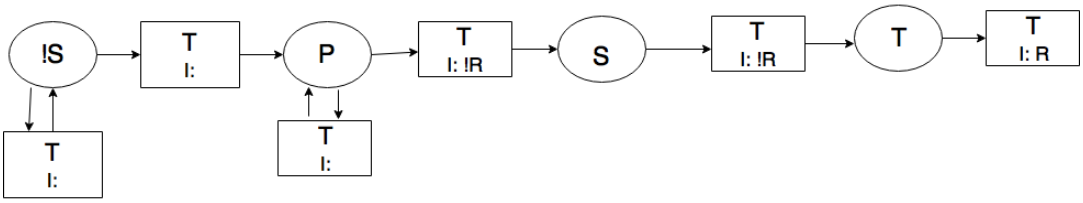


Figure B.47.: Mapping LTL to FOL conversion - Precedence Chain II 'before R' [WMS+19]

After Q

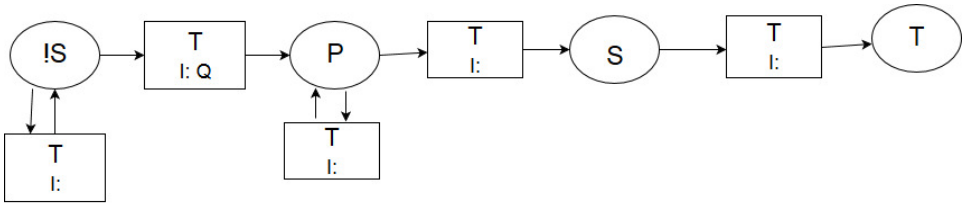


Figure B.48.: Mapping LTL to FOL conversion - Precedence Chain II 'After Q' [WMS+19]

Between Q and R

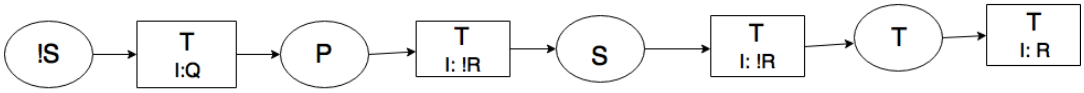


Figure B.49.: Mapping LTL to FOL conversion - Preced. Chain II 'Between Q and R' [WMS+19]

After Q until R

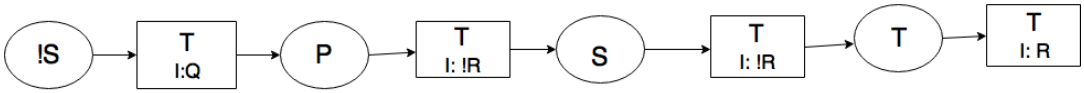


Figure B.50.: Mapping LTL to FOL conversion - Preced. Chain II 'After Q until R' [WMS+19]

C. Publications

In this work, a formalization process for test and requirements data was shown. This was done in a step-wise qualitative form as well as with quantitative case studies. For all formalization steps, qualitative and quantitative examples, a primary publication by the author of this work exists. This section provides an overview of these publications with a short description and connection to the other publications.

[WHPR17] ***Walter et al. - "A Formalization Method to Process Structured Natural Language to Logic Expressions to Detect Redundant Specification and Test Statements"***
In this paper, the formalization process chain (see Figure 3.1) is introduced with examples of test cases. A qualitative study shows redundancy checks for test steps as one industrial use case.

[WSPR18] ***Walter et al. - "Improving Test Execution Efficiency Through Clustering and Reordering of Independent Test Steps"***
The former introduced redundancy check requires post processing for industrial use. This is shown with this work. Existing test sets are checked for redundancies and are rearranged towards an efficient, redundancy-free test set.

[WMR18] ***Walter et al. - "A Method to Automatically Derive the System State Machine from Structured Natural Language Requirements through Requirements Formalization"***
In this paper, the formalization process chain is applied to requirements. The formalization is extended so that a consistent system state machine including all requirements is derived.

[WMS+19] ***Walter et al. - "Executable State Machines Derived from Structured Textual Requirements - Connecting Requirements and Formal System Design"***
The former state machine representation is extended with an execution layer that allows the user to interact and control the state machine through a GUI with input and to observe the system reaction through its output.

[WKR18] ***Walter et al. - "Machine-Executable Model-Based Systems Engineering with Graph-Based Design Languages"***
This abstract is the summary of a poster that envisions a fully digital development process. It introduces the term "Machine-Executable V-model'

[WKR19] ***Walter et al. - "From Manual to Machine-Executable Model-based Systems Engineering via Graph-based Design Languages"***
This paper represents the extension of paper [WKR18] from abstract to full paper. It elaborates on the possibilities of digital development and showcases with qualitative examples how an 'Machine-Executable V-model' can be envisioned

D. Bibliography

[AF00] Alessandro Artale and Enrico Franconi. A Survey of Temporal Extensions of Description Logics. *Annals of Mathematics and Artificial Intelligence*, 30:pages 171–210, 2000.

[Ale85] Christopher Alexander. The Timeless Way of Building. *Alexander, The Production of Houses*, 1985.

[AR03] Rolf Alber and Stephan Rudolph. '43'— A Generic Approach for Engineering Design Grammars. In *Proceedings AAAI Spring Symposium "Computational Synthesis," Stanford, CA, AAAI Technical Report SS-03-02*, 2003.

[AR04] Rolf Alber and Stephan Rudolph. On a grammar-based design language that supports automated design generation and creativity. In *Knowledge Intensive Design Technology*, pages 19–35. Springer, 2004.

[Art10] Alessandro Artale. *Formal Methods - Linear Temporal Logic*. University of Bolzano, 2010. Lecture Notes.

[Ben18] Bengaluru. Department of Computer Science and Automation, https://drona.csa.iisc.ernet.in/ deepakd/atc-2011/lba.pdfl. Jan 2018.

[BN13] Eberhard Bergmann and Helga Noll. *Mathematische Logik mit Informatik-Anwendungen*, volume 187. Springer-Verlag, 2013.

[BW07] Brian Berenbach and Timo Wolf. A Unified Requirements Model; Integrating Features, Use Cases, Requirements, Requirements Analysis and Hazard Analysis. In *Global Software Engineering, 2007. ICGSE 2007. Second IEEE International Conference on*, pages 197–203. IEEE, 2007.

[CAM18] University of Cambridge - Computer Laboratory, https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/turing-machine/one.html. Jan 2018.

[CB98] IEEE Computer Society. Software Engineering Standards Committee and IEEE-SA Standards Board. Recommended Practice for Software Requirements Specifications. Institute of Electrical and Electronics Engineers, 1998.

[Cer80] Eduard Cerny. Characteristic Functions in Multivalued Logic Systems. *DIGITAL PROC*, 6(2):pages 167–174, 1980.

[Cla96] Peter Clark. Requirements for a Knowledge Representation System. *www.cs.utexas.edu/users/pclark/working_notes/010.pdf*, 1996.

[Col17] Collinsdictionary. Definition of 'inference', https://www.collinsdictionary.com/dictionary/english/inference. 2017.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-state Verification. In *Proceedings of the second workshop on formal methods in software practice*, pages 7–15. ACM, 1998.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 411–420. IEEE, 1999.

[DAC17] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. A Specification Pattern System, http://people.cs.ksu.edu/scriptstyle/mathtt/sim/dwyer/SPAT/ltl.html. 2017.

[Dam96] Peter Damerow. Abstraction and Representation. In *Abstraction and Representation*, pages 371–381. Springer, 1996.

[Dar68] Charles Darwin. On the Origin of Species by Means of Natural Selection. 1859. *London: Murray Google Scholar*, 1968.

[DG97] Marco Dorigo and Luca Maria Gambardella. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on evolutionary computation*, 1(1):pages 53–66, 1997.

[DP117] Sourcemaking Design Patterns, https://sourcemaking.com/design-patterns. Sep 2017.

[EO117] English Oxford Dictionaries, https://en.oxforddictionaries.com. Aug 2017.

[FHDW15] Felix Föcker, Frank Houdek, Marian Daun, and Thorsten Weyer. Model-based Engineering of an Automotive Adaptive Exterior Lighting System: Realistic Example Specifications of Behavioral Requirements and Fnctional Design. Technical report, ICB-Research Report, 2015.

[Fra15] Aldrin Jaramillo Franco. Requirements Elicitation Approaches: A Systematic Review. In *9th International Conference on Research Challenges in Information Science (RCIS)*, pages 520–521. IEEE, 2015.

[GG16] Valentin Goranko and Antony Galton. Temporal Logic, Winter Edition 2015, The Stanford Encyclopedia of Philosophy, Edward N. Zalta (ed.). Jul 2016. URL = http://plato.stanford.edu/archives/win2015/entries/logic-temporal/.

[Gli07] Martin Glinz. On Non-functional Requirements. In *15th IEEE International Requirements Engineering Conference*, pages 21–26. IEEE, 2007.

[Gmb16] Sophist GmbH. *"MASTeR" - Schablonen für alle Fälle*, volume 3. www.sophist.de, 2016.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *International Conference on Computer-Aided Verification*, pages 53–65. Springer, 2001.

[GS80] James Gips and George Stiny. Production Systems and Grammars: A Uniform Characterization. *Environment and Planning B: Planning and Design*, 7(4):pages 399–408, 1980.

[Hey99] Francis Heylighen. The Growth of Structural and Functional Complexity During Evolution. *The Evolution of Complexity*, pages 17–44, 1999.

[HKS+10] Jonas Helming, Maximilian Koegel, Florian Schneider, Michael Haeger, Christine Kaminski, Bernd Bruegge, and Brian Berenbach. Towards a Unified Requirements Modeling Language. In *Fifth International Workshop on Requirements Engineering Visualization (REV)*, pages 53–57. IEEE, 2010.

[Hoo94] Ivy Hooks. Writing good requirements. In *INCOSE International Symposium*, volume 4, pages 1247–1253. Wiley Online Library, 1994.

[IEE90] Computer-Society IEEE. IEEE Standard Glossary of Software Engineering Terminology. 1990.

[IIL17a] IILS mbh. Design Cockpit 43, https://www.iils.de. 2017.

[IIL17b] IILS mbh. https://www.iils.de/downloads/IILS-WhitePaper-TotalEngineeringAutomation.pdf. 2017.

[ISO05] EN ISO. 9000: 2005. *Quality management systems-Fundamentals and vocabulary (ISO 9000: 2005)*, 2005.

[ISO11] *ISO/IEC/IEEE 29148: 2011(E): ISO/IEC/IEEE International Standard - Systems and Software Engineering - Life Cycle Processes - Requirements Engineering*. IEEE, 2011.

[ISTQB15] ISTQB International Software Testing Qualifications Board. *Standard Glossary of Terms used in Software Testing Version 3.1*. 2015.

[Jac17] Michael Jackson. The Right-hand Side Problem: Research topics in RE. In *25th International Requirements Engineering Conference (RE)*, pages 474–475. IEEE, 2017.

[Kam68] Johan Anthony Wilem Kamp. Tense Logic and the Theory of Linear Order. 1968.

[KC02] Sascha Konrad and Betty HC Cheng. Requirements Patterns for Embedded Systems. In *Joint International Conference on Requirements Engineering*, pages 127–136. IEEE, 2002.

[KC05a] Sascha Konrad and Betty HC Cheng. Automated Analysis of Natural Language Properties for UML Models. In *International Conference on Model Driven Engineering Languages and Systems*, pages 48–57. Springer, 2005.

[KC05b] Sascha Konrad and Betty HC Cheng. Real-time Specification Patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381. ACM, 2005.

[KVBSV12] Timothy Kam, Tiziano Villa, Robert K Brayton, and Alberto L Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Logic Optimization*. Springer Science & Business Media, 2012.

[Lau08] Søren Lauesen. *Guide to Requirements SL-07-Template with Examples*. Lauesen Publishing, 2008.

[LL12] Xinye Lu and Guiming Luo. Direct Translation of LTL Formulas to Büchi Automata. In *11th International Conference on Cognitive Informatics and Cognitive Computing (ICCI CC)*, pages 323–328. IEEE, 2012.

[LNHK11] Yang Li, Nitesh Narayan, Jonas Helming, and Maximilian Koegel. A Domain-Specific Requirements Model for Scientific Computing (NIER Track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 848–851. ACM, 2011.

[Mar18] Jan Martin. Automatisches Generieren von Finite State Machines aus Structured Natural Language Requirements, 2018.

[Mav12] Alistair Mavin. Listen, then use EARS. *IEEE software*, 29(2):17–18, 2012.

[MB07] Mercedes-Benz. *Requirements Guideline*. 2007.

[MB17] Mercedes-Benz. *Test Specification Guideline*. 2017.

[MT13] Walid Maalej and Anil Kumar Thurimella. *Managing Requirements Knowledge*. Springer, 2013.

[MW10] Alistair Mavin and Philip Wilkinson. Big Ears (The Return of "Easy Approach to Requirements Engineering"). In *18th IEEE International Requirements Engineering Conference (RE)*, pages 277–282. IEEE, 2010.

[MWGU16] Alistair Mavin, Philip Wilksinson, Sarah Gregory, and Eero Uusitalo. Listens Learned (8 Lessons Learned Applying EARS). In *24th International Requirements Engineering Conference (RE)*, pages 276–282. IEEE, 2016.

[MWHN09] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. Easy Approach to Requirements Ryntax (EARS). In *17th International Requirements Engineering Conference (RE)*, pages 317–322. IEEE, 2009.

[NK98] Klaus North and Gita Kumta. *Wissensorientierte Unternehmensführung*. Springer, 1998.

[PBFG14] Gerhard Pahl, Wolfgang Beitz, Jörg Feldhusen, and Karl-Heinrich Grote. *Pahl/Beitz Konstruktionslehre: Grundlagen erfolgreicher Produktentwicklung. Methoden und Anwendung.* Springer-Verlag, 2014.

[PBSJ13] Gerhard Pahl, Wolfgang Beitz, Hans-Joachim Schulz, and U Jarecki. *Pahl/Beitz Konstruktionslehre: Grundlagen erfolgreicher Produktentwicklung. Methoden und Anwendung.* Springer-Verlag, 2013.

[PH03] Arthur N Prior and Per FV Hasle. *Papers on Time and Tense.* Oxford University Press on Demand, 2003.

[PH12] Amalinda Post and Jochen Hoenicke. Formalization and Analysis of Real-time Requirements: A Feasibility Study at Bosch. In *International Conference on Verified Software: Tools, Theories, Experiments*, pages 225–240. Springer, 2012.

[PH13] Przemyslaw Prusinkiewicz and James Hanan. *Lindenmayer Systems, Fractals, and Plants*, volume 79. Springer Science & Business Media, 2013.

[PHP11] Amalinda Post, Jochen Hoenicke, and Andreas Podelski. Vacuous Real-time Requirements. In *19th International Requirements Engineering Conference (RE)*, pages 153–162. IEEE, 2011.

[Pia52] J Piaget. The Child's Conception of Numbers (C. Gattegno & FM Hodgson, Trans.), 1952.

[PL12] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants.* Springer Science & Business Media, 2012.

[PMHP12] Amalinda Post, Igor Menzel, Jochen Hoenicke, and Andreas Podelski. Automotive Behavioral Requirements Expressed in a Specification Pattern System: a Case Study at Bosch. *Requirements Engineering*, 17(1):19–33, 2012.

[PMP11] Amalinda Post, Igor Menzel, and Andreas Podelski. Applying Restricted English Grammar on Automotive Requirements — Does it Work? A Case Study. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 166–180. Springer, 2011.

[Pri67] Arthur N Prior. *Past, Present and Future*, volume 154. Clarendon Press Oxford, 1967.

[Pri03] Arthur N Prior. *Time and Modality*. John Locke Lecture, 2003.

[RP15] Chris Rupp and Klaus Pohl. *Basiswissen Requirements Engineering: Aus- und Weiterbildung nach IREB-Standard zum Certified Professional for Requirements Engineering Foundation Level*. dpunkt. verlag, 2015.

[RR00] James Robertson and Suzanne Robertson. Volere. *Requirements Specification Templates*, 2000.

[Rud02] Stephan Rudolph. *Übertragung von Ähnlichkeitsbegriffen*. Habilitation, Institut für Statik und Dynamik der Luft- und Raumfahrtkonstruktionen, Universität Stuttgart, 2002.

[Rud06] Stephan Rudolph. A Semantic Validation Scheme for Graph-based Engineering Design Grammars. In *Design Computing and Cognition*, pages 541–560. Springer, 2006.

[SA95] Robert Shishko and Robert Aster. NASA Systems Engineering Handbook. *NASA Special Publication*, 6105, 1995.

[SH18] Christian Schwarzl and Jens Herrmann. Systematic Test Platform Selection: Reducing Costs for Testing Software-Based Automotive E/E Systems. In *11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 374–383. IEEE, 2018.

[SL12] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest*. dpunkt, 2012.

[SPZ12] Liwei Shen, Xin Peng, and Wenyun Zhao. Software Product Line Engineering for Developing Self-Adaptive Systems: Towards the Domain Requirements. In *36th Annual Computer Software and Applications Conference (COMPSAC)*, pages 289–296. IEEE, 2012.

[SR05] Jörg Schäfer and Stephan Rudolph. Satellite Design by Design Grammars. *Aerospace Science and Technology*, 9(1):pages 81–91, 2005.

[SR16] Jens Schmidt and Stephan Rudolph. Graph-Based Design Languages: A Lingua Franca for Product Design Including Abstract Geometry. *Computer Graphics and Applications*, 36(5):88–93, 2016.

[ST118] StandfordEdu, https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.htm. Jan 2018.

[Sti80] George Stiny. Introduction to Shape and Shape Grammars. *Environment and Planning B: Planning and Design*, 7(3):pages 343–351, 1980.

[Thi99] Robert J Thierauf. *Knowledge Management Systems for Business*. Greenwood Publishing Group, 1999.

[VIR10] University of Virgina - Theory of Computation. Feb 2010.

[VKBSV13] Tiziano Villa, Timothy Kam, Robert K Brayton, and Alberto L Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Functional Optimization*. Springer Science & Business Media, 2013.

[WDR11] Md Tawhid Bin Waez, Juergen Dingel, and Karen Rudie. Timed Automata for the Development of Real-time Systems. *Research Report 2011–579*, 2011.

[WHPR17] Benedikt Walter, Jakob Hammes, Marco Piechotta, and Stephan Rudolph. A Formalization Method to Process Structured Natural Language to Logic Expressions to Detect Redundant Specification and Test Statements. In *IEEE 25th International Requirements Engineering Conference (RE)*, pages 263–272, 2017.

[Wie99] Karl E Wiegers. Writing Quality Requirements. *Software Development*, 7(5):pages 44–48, 1999.

[WKR18] Benedikt Walter, Dennis Kaiser, and Stephan Rudolph. Machine-Executable Model-Based Systems Engineering via Graph-Based Design Languages. In *Complex Systems Design & Management. In Conference proceedings CSDM*, page 239. Springer, 2018.

[WKR19] Benedikt Walter, Dennis Kaiser, and Stephan Rudolph. From Manual to Machine-Executable Model-Based Systems Engineering via Graph-Based Design Languages. In *7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 203–210, 2019.

[WMR17] Lenis R Wong, David S Mauricio, and Glen D Rodriguez. A Systematic Literature Review about Software Requirements Elicitation. *Journal of Engineering Science and Technology*, 12(2):296–317, 2017.

[WMR18] Benedikt Walter, Jan Martin, and Stephan Rudolph. A Method to Automatically Derive the System State Machine from Structured Natural Language Requirements through Requirements Formalization. In *Incose EMEASEC Conference, Tag des Systems Engineerings (TdSE)*, 2018.

[WMS+19] Benedikt Walter, Jan Martin, Jonathan Schmidt, Hanna Dettki, and Stephan Rudolph. Executable State Machines Derived from Structured Textual Requirements - Connecting Requirements and Formal System Design. In *7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 195–202, 2019.

[WSPR18] Benedikt Walter, Maximilian Schilling, Marco Piechotta, and Stephan Rudolph. Improving Test Execution Efficiency Through Clustering and Reordering of Independent Test Steps. In *IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 363–373, 2018.

[ZJ97] Pamela Zave and Michael Jackson. Four Dark Corners of Requirements Engineering. *Transactions on Software Engineering and Methodology (TOSEM)*, 6(1):pages 1–30, 1997.

E. Curriculum Vitae

Benedikt Walter

Personal Data

Born on 25^{th} of August 1988 in Ellwangen (Jagst)
Parents: Ulrich and Andrea Walter
Marital Status: married to Christina Colondres Walter
Current Address: Kriegswiesenstrasse 14, 72131 Ofterdingen, Germany

Education

University of Stuttgart, DE — Oct 2013 - Jun 2015
Master of Science, Aerospace Engineering

North Carolina State University, US — Aug 2011 - Jun 2012
Study Abroad

University of Stuttgart, DE — Oct 2009 - Sep 2013
Bachelor of Science, Aerospace Engineering

Quenstedt Gymnasium Mössingen, DE — Sep 1999 - Jun 2008
German Abitur

Work Experience

Mercedes-Benz, Daimler AG — May 2018 - today
Development Engineer, Highly Automated Driving

Mercedes-Benz, Daimler AG
PhD Student, E/E System Integration, Requirements Validation — May 2017 - Apr 2018
PhD Student, E/E System Integration, Testing Process — Jul 2015 - Apr 2017