

# Test Case Generation from Natural Language Requirements for Embedded Systems with Semantic Role Labeling

Alexander Kugler

Department of Computer Science

Technical Report

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

<http://aib.informatik.rwth-aachen.de/>

# **Test Case Generation from Natural Language Requirements for Embedded Systems with Semantic Role Labeling**

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der  
RWTH Aachen University zur Erlangung des akademischen Grades  
eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von

**M. Sc. RWTH**  
**Alexander Kugler**  
aus Aachen

Berichter: Prof. Dr.-Ing. Stefan Kowalewski  
Prof. Dr. rer. nat. Bernhard Rumpe

Tag der mündlichen Prüfung: 7. Januar 2025

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.



Alexander Kugler  
Lehrstuhl Informatik 11  
alexander.kugler@rwth-aachen.de

---

Aachener Informatik Bericht AIB-2025-04

Herausgeber: Fachgruppe Informatik  
RWTH Aachen University  
Ahornstr. 55  
52074 Aachen  
GERMANY

ISSN 0935-3232



## Abstract

The work presented in this thesis explores the application of Semantic Role Labeling (SRL) for the generation of test cases from natural language requirements for embedded systems. The approach, labelled Test Generation with Semantic Role Labeling (TG-SRL), is composed of five stages and combines machine learning with a rule-based approach. Information extracted via Semantic Role Labeling (SRL) is initially aggregated into logical expressions before being translated into First-Order Logic (FOL) formulae. Test case generation is achieved using Satisfiability Modulo Theory (SMT) solving. By modifying the SMT instance according to defined tactics, a test suite is generated.

The thesis concludes with an evaluation of TG-SRL using a mutant-based strength analysis, and a comparison with the *Nat2Test* approach from Carvalho et al. TG-SRL performs favorably and provides valuable insights into employing Natural Language Processing (NLP) methods, and in particular SRL, in the field of test case generation. The methods and concepts presented in this thesis have been implemented in a publicly available research framework.





## Zusammenfassung

Die in dieser Dissertation vorgestellte Arbeit untersucht den Einsatz von "Semantic Role Labeling (SRL)" zur Generierung von Testfällen aus natürlichsprachlichen Anforderungen für eingebettete Systeme. Der Ansatz, der als "Test Generation with Semantic Role Labeling (TG-SRL)" bezeichnet wird, besteht aus fünf Phasen und kombiniert maschinelles Lernen mit einem regelbasierten Ansatz. Die mittels SRL extrahierten Informationen werden zunächst zu logischen Ausdrücken aggregiert, bevor sie in Formeln der Prädikatenlogik erster Ordnung (FOL) übersetzt werden. Die Generierung von Testfällen erfolgt durch die Lösung von Erfüllbarkeits-Modulo-Theorie (SMT) Instanzen. Durch die Modifikation der SMT-Instanzen gemäß definierten Taktiken wird eine Testsuite erzeugt.

Die Dissertation schließt mit einer Evaluierung von TG-SRL mittels einer auf Mutanten basierenden Stärkeanalyse und einem Vergleich mit dem Ansatz *Nat2Test* von Carvalho et al. ab. TG-SRL zeigt eine vorteilhafte Performance und liefert wertvolle Einblicke in den Einsatz von Methoden des Natural Language Processing (NLP), insbesondere von SRL, im Bereich der Testfallgenerierung. Die in dieser Arbeit vorgestellten Methoden und Konzepte wurden in einem öffentlich zugänglichen Forschungsframework implementiert.



## Vorwort

Die vorliegende Dissertation ist während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Informatik 11 - Embedded Software der Rheinisch-Westfälischen Technischen Hochschule Aachen entstanden.

Herrn Prof. Dr.-Ing. Stefan Kowalewski, Leiter des Lehrstuhls Informatik 11 der RWTH Aachen, danke ich für die intensive Förderung und Betreuung meiner Arbeit, den interessanten technischen Austausch über die letzten Jahre hinweg, und für seine Tätigkeit als Berichterstatter im Rahmen der Promotionsprüfung. Ebenfalls danke ich Herrn Prof. Dr. rer. nat. Bernhard Rumpe für die Übernahme des Korreferats, Frau Prof. Dr.-Ing. Ulrike Meyer für die Übernahme der Fachprüfung und Herrn Prof. Dr. rer. nat. Matthias Müller für den Vorsitz in der Prüfungskommission.

Desweiteren gilt mein Dank meinen Kollegen am Lehrstuhl für Informatik 11, die stets zu einem konstruktiven und freundlichen Arbeitsklima beigetragen haben und immer offen für einen fachlichen Austausch waren. Weiterhin danke ich all denjenigen, die bei der Korrektur dieser Arbeit engagierte Unterstützung geleistet haben.

Abschließend gilt mein besonderer Dank meiner Familie und meinem näheren persönlichen Umfeld für die fortwährende moralische Unterstützung und das entgegengebrachte Verständnis während jeder Phase der Erstellung dieser Arbeit. Meinen Eltern danke ich vor allem für Ihre Förderung und bedingungslose Unterstützung während meiner gesamten akademischen Laufbahn.

Aachen, 06. April 2025

Alexander Kugler



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Scope, Challenges and Objectives . . . . .	2
1.2	Solution Approach & Contributions . . . . .	3
1.3	Bibliographic Notes . . . . .	4
1.4	Outline . . . . .	5
<b>2</b>	<b>Fundamentals</b>	<b>7</b>
2.1	Development of Embedded Software in the Automotive Industry . . . . .	7
2.1.1	A-SPICE & Autosar Classic . . . . .	7
2.1.2	Verification & Validation Activities . . . . .	10
2.2	Test Generation from Requirements . . . . .	11
2.2.1	Model-Based Test Generation . . . . .	12
2.2.2	Mutant-Based Strength Analysis . . . . .	14
2.3	Natural Language Processing . . . . .	14
2.3.1	NLP Methods and Common Tasks . . . . .	15
2.3.2	Semantic Role Labeling with Proposition Bank Annotations . . . . .	17
2.3.3	Frame Semantic Parsing . . . . .	20
2.3.4	NLP Frameworks . . . . .	22
<b>3</b>	<b>Related Work</b>	<b>25</b>
3.1	Natural Language as the Input Domain . . . . .	25
3.1.1	Solimva . . . . .	25
3.1.2	Litmus . . . . .	26
3.1.3	Retna . . . . .	27
3.2	Use Cases as the Input Domain . . . . .	28
3.2.1	UMTS . . . . .	28
3.2.2	Text2Test . . . . .	29
3.2.3	Test Case Generation, Selection and Coverage from Natural Language . . . . .	29
3.3	Controlled Natural Language as the Input Domain . . . . .	30
3.3.1	Generating Test Cases for Timed Systems from CNL Specifications . . . . .	30
3.3.2	Nat2Test . . . . .	31
<b>4</b>	<b>Test Generation with Semantic Role Labeling</b>	<b>35</b>
4.1	Overview . . . . .	35
4.2	Stage 1: Signal Attributes & Preprocessing of Requirements . . . . .	38
4.3	Stage 2: Information Extraction via NLP . . . . .	39
4.3.1	Requirement & Predicate Frames . . . . .	41

4.4	Stage 3: Processing and Aggregation of Extracted Information into Logical Expressions . . . . .	46
4.4.1	Enriching Predicate Frames . . . . .	47
4.4.2	Formalization into Logical Expressions . . . . .	49
4.4.3	Atomic Expressions for Conditions and Actions . . . . .	55
4.5	Stage 4: Formalization into First-Order Logic . . . . .	58
4.5.1	Test Case Generation . . . . .	59
4.6	Stage 5: Test Suite Generation . . . . .	64
4.7	Implementation . . . . .	67
4.7.1	Frameworks and Tools . . . . .	69
4.7.2	Preprocessing of Requirements . . . . .	70
<b>5</b>	<b>Evaluation</b>	<b>75</b>
5.1	Window Control System . . . . .	76
5.1.1	Mutant-Based Strength Analysis . . . . .	77
5.2	Vending Machine System . . . . .	81
5.2.1	Mutant-Based Strength Analysis . . . . .	85
5.3	Daimler Turn Indicator System . . . . .	91
5.3.1	Mutant-Based Strength Analysis . . . . .	93
5.4	Evaluation of Objectives . . . . .	94
5.5	Limitations & Future Work . . . . .	95
5.5.1	Large Language Models . . . . .	96
<b>6</b>	<b>Conclusion</b>	<b>101</b>
<b>7</b>	<b>Appendix</b>	<b>103</b>
	<b>Bibliography</b>	<b>136</b>

## List of Tables

2.1	Exemplary signal specified as time-value pairs. Step size of 0.1. . . . .	13
2.2	Overview of core Proposition Bank (ProbBank) semantic roles. . . . .	19
2.3	Semantic roles of commas according to [42]. . . . .	21
2.4	ProbBank Framesets for "to buy" and "to sell". . . . .	22
3.1	Exemplary requirement, taken from [68], conforming to the grammar from Listing 3.1. . . . .	31
3.2	Mutant-based strength analysis results from [8]: Nat2Test compared to random testing. . . . .	33
4.1	Content of a requirement frame. . . . .	41
4.2	Content of a predicate frame. . . . .	42
4.3	Exemplary predicate frame based on SRL output in Figure 4.1. . . . .	44
4.4	Possible rephrasings for the given sentence. . . . .	44
4.5	Exemplary requirement frame with identified conditional parts. . . . .	45
4.6	Overview of inspected tags and examples for associated regular expressions. . . . .	48
4.7	Exemplary predicate frames. . . . .	49
4.8	Exemplary predicate frames. . . . .	50
4.9	Expressions produced by the "extended" tactic for the disjunction $x \vee y \vee z \vee u$ . . . . .	67
4.10	Overview of disjunctions generated with the "normal" comparison operator tactic. . . . .	67
5.1	Input and output signals from the <i>Window Control</i> system. . . . .	79
5.2	Mutation testing results for the <i>Window Control</i> system. . . . .	80
5.3	Input and output signals from the <i>Vending Machine</i> system. . . . .	83
5.4	Enumeration to integer mapping for the <i>Vending Machine</i> system. . . . .	83
5.5	Enumeration in integer mapping for the <i>Vending Machine</i> system. . . . .	84
5.6	Exemplary test case from <i>Nat2Test</i> for the <i>Vending Machine</i> system. . . . .	84
5.7	Excerpt of an exemplary test case from TG-SRL for the <i>Vending Machine</i> system. . . . .	85
5.8	Mutation testing results for the <i>Vending Machine</i> system. . . . .	85
5.9	Illustration why the condition "timer $\geq 10$ && timer $\leq 30$ && mode == 3" will always be triggered with a timer value of 10. . . . .	88
5.10	Mutation testing results for the <i>Vending Machine</i> system after accounting for semantically equivalent mutants. . . . .	89
5.11	Mutation testing results for the <i>Vending Machine</i> system after enforcing changes of the initial value for a signal. . . . .	90
5.12	Mutation testing results for the <i>Vending Machine</i> system. . . . .	91

## List of Tables

5.13	Input and output signals from the <i>Turn Indicator</i> system. . . . .	92
5.14	Mutation testing results for the <i>Turn Indicator</i> system. . . . .	93
7.1	List of requirements for the <i>Window Control</i> system. . . . .	104
7.2	Mutations of the <i>Window Control</i> system . . . . .	106
7.3	Preprocessed list of requirements for the <i>Vending Machine</i> system. . . . .	106
7.4	List of requirements for the <i>Vending Machine</i> system in the SysReq-CNL. . . . .	107
7.5	Preprocessed list of requirements for the <i>Turn Indicator</i> system. . . . .	113
7.7	Exemplary test case from TG-SRL for the <i>Vending Machine</i> system. . . . .	126
7.6	Output from [46] on the sentence "If a coin is inserted while the machine is in state 1 and the request timer is lower or equal to 30, then the machine state should be modified to 3 and the request timer should be reset." . . .	128



## List of Figures

1.1	Overview of the TG-SRL approach presented in this thesis. . . . .	4
2.1	Overview of Automotive SPICE (A-SPICE), adapted from [5]. . . . .	8
2.2	Autosar Classic architecture and communication between layers [13]. . . .	9
2.3	Application software structure according to Autosar Classic. . . . .	10
2.4	Simplified overview of model-based test generation. . . . .	12
2.5	Dependency parse trees for two exemplary sentences. . . . .	17
2.6	Semantic Role Labeling output from the <i>AllenNLP</i> demo [41]. . . . .	18
2.7	Semantic Role Labeling output from the <i>AllenNLP</i> demo [41]. . . . .	18
2.8	Framesets from ProbBank for the word "decline". . . . .	20
2.9	Frame Semantic Parsing example from Swayamdipta et al. [44]. . . . .	21
3.1	Overview of <i>Solimva</i> based on a figure in [60]. . . . .	26
3.2	Discourse Representation Structures (DRS) example from [61]. . . . .	27
3.3	Simplified case frame example from Table 1 in [8]. . . . .	32
4.1	Exemplary SRL tags created with the <i>AllenNLP</i> online demo [41]. . . . .	43
4.2	Possible interpretations of the temporal behavior for the requirement "If <i>signal a</i> is 2 for 3 seconds, then add 10 to <i>signal b</i> after 5 seconds.". . . .	52
4.3	SRL output from <i>CogComp-NLP</i> [76] showcasing valid relation spans. . . .	53
4.4	SRL output from <i>CogComp-NLP</i> [76] showcasing overlapping and thus invalid relation spans. . . . .	53
4.5	Exemplary output signal with linear and constant behavior. . . . .	61
4.6	Exemplary output from <i>CogComp-NLP</i> [76]. . . . .	72
4.7	Exemplary output from <i>AllenNLP</i> [41]. . . . .	73
4.8	A comma placed before the "and" results in erroneous output. Example from <i>AllenNLP</i> [41]. . . . .	74
5.1	<i>MuJava</i> method-level mutation example. . . . .	76
5.2	Black-box view on the <i>Window Control</i> system. . . . .	77
5.3	Detailed view on the <i>Window Control</i> system. . . . .	78
5.4	Variant subsystems for the <i>Window Control</i> system to facilitate mutation testing. . . . .	80
5.5	State machine for the <i>Vending Machine</i> system. Conditions are colored in blue while actions are colored in red. . . . .	82
5.6	SRL-Verb, SRL-Nom and SRL-Prep output from [76]. . . . .	98
5.7	Generated test cases from GPT-4o [89] on the given input requirement. . .	99

*List of Figures*

5.8	Example that illustrates how GPT-4o [89] can be utilized to resolve ambiguity in a given requirement. . . . .	100
-----	---	-----

## List of Acronyms

A-SPICE	Automotive SPICE
BERT	Bidirectional Encoder Representations from Transformers
CAN	Controller Area Network
CASE	Connected, Autonomous, Shared and Electric
CNL	Controlled Natural Language
CoNLL	Conference on Computational Natural Language Learning
CSP	Communicating Sequential Processes
DFRS	Data-Flow Reactive Systems
DRS	Discourse Representation Structures
FOL	First-Order Logic
FSP	Frame Semantic Parsing
IMR	Internal Model Representation
LIN	Local Interconnect Network
LLM	Large Language Models
LLM	Large Language Model
MBD	Model-based Development
MBSE	Model-based System Engineering
NER	Named Entity Recognition
NLP	Natural Language Processing
NLU	Natural Language Understanding
OCL	Object Constraint Language
POS	Part of Speech
ProbBank	Proposition Bank
RTE	Runtime Environment

## *List of Acronyms*

RUCM	Restricted Use Case Modeling
RWTH Aachen	Rheinisch-Westfälische Technische Hochschule Aachen
SCR	Software Cost Reduction
SDV	Software Defined Vehicle
SMT	Satisfiability Modulo Theory
SRL	Semantic Role Labeling
SuT	System under Test
SW-C	Software Component
TG-SRL	Test Generation with Semantic Role Labeling
UML	Unified Modeling Language
V&V	Validation & Verification
WSD	Word Sense Disambiguation

# 1 Introduction

In the automotive industry there is a paradigm change towards Software Defined Vehicles (SDVs). Connected, Autonomous, Shared and Electric (CASE) mobility is disrupting the automotive value chain, and software is emerging as the principal driver of innovation [1]. Comparable to today's smartphone, a SDV needs to be seamlessly integrated into a large software ecosystem. This necessitates regular software deployments via over-the-air updates to close security vulnerabilities, improve existing features, or simply deliver new ones. However, in contrast to software for smartphones, software in a vehicle needs to adhere to strict quality standards and safety regulations, such as the ISO26262 [2]. Thus, rigorous testing is critical to guarantee software quality and adherence to regulations. Nowadays, testing efforts can contribute between 40-70 percent of the overall project costs [3]. Without countermeasures, this share can be expected to increase significantly in the context of SDVs.

A core strategy to tackle the challenge of increasing update frequency and rising testing efforts is automation. Efforts for testing software can be split into effort for test planning, design, implementation, execution and evaluation. The focus in this thesis is on the design and implementation phase, i.e., it is assumed that a test strategy has been derived and requirements-based testing has been identified as a required activity. In today's automotive industry, the design and implementation of requirements-based test cases is either partially automated or a manual activity. In cases of partial automation, most approaches entail significant manual effort [4]. Thus, the automation of test generation has significant potential to improve efficiency in software development and reduce associated costs. While there exist a few approaches which enable test case generation based on formal models, the effort to create such formal models out of requirements can be considerable. In many cases, effort for specifying test cases directly from natural language requirements simply shifts to the formalization of these requirements. In contrast, generating requirements-based tests directly from natural language requirements has proven difficult due to the ambiguity, incompleteness, and sometimes inconsistency of such requirements.

This thesis proposes an approach that tries to tackle these challenges with the help of Natural Language Processing (NLP) methods. In particular, a framework for test generation based on natural language requirements using Semantic Role Labeling (SRL) is presented. In addition, the feasibility and performance of the proposed approach is investigated. As an acronym for the proposed method, TG-SRL (**T**est **G**eneration with **S**emantic **R**ole **L**abeling) is employed throughout this thesis.

The remainder of this chapter is structured as follows. First, the scope, problem statement, and objectives of this thesis are defined. Next, the chosen solution approach is summarized, and major contributions are highlighted. At the end of the chapter, an outline for the remaining thesis is presented.

### 1.1 Scope, Challenges and Objectives

This thesis focuses on automatic test generation based on natural language requirements for embedded systems. It is assumed that the requirements intended for test generation have a sufficient level of detail for the generation of executable test cases. The requirements are also assumed to describe a System under Test (SuT) as a black-box by specifying system behavior based on observable in- and output signals. Applied to the automotive context, where software is often developed according to the A-SPICE standard [5], the presented approach is applicable to requirements that are used in the process areas SWE.1 to SWE.3 (see Section 2.1.1).

Furthermore, the thesis is scoped to functional testing, also referred to as requirements-based or specification-based testing. In general, it is possible to distinguish between structural and functional testing (see Section 2.1.2). While structural tests can be generated automatically based on the structure of the System under Test (SuT), they do not necessarily reveal errors in the functional behavior of software. Functional testing instead directly tests the behavior of the SuT against the expected behavior that has previously been specified in the requirements. Regulations in the automotive domain, such as the ISO26262-6 [2], mandate functional testing for safety critical software. Generally, this test category is crucial for delivering a reliable product with a satisfactory and dependable user experience.

While solutions exist to generate functional test cases based on formalized requirements, requirements are usually specified in natural language [6]: according to surveys, 79 % of requirements are documented in free-flow natural language [3] and only 7 % use a formal specification [7]. Although natural language is inherently ambiguous, it is used for ease in comprehension and ease in sharing between the different stakeholders [7]. Existing approaches in literature to generate test cases based on natural language requirements face several major challenges:

#### Challenges

- **Input domain restrictions:** The input domain is restricted in many approaches to reduce or eliminate challenges that come with natural language. For example, some methods enforce the adherence to a specific template, while others employ a Controlled Natural Language (CNL) defined by a grammar. Restrictions on the input domain can significantly simplify information extraction, and the formalization into a representation that enables automatic test generation. The trade-off, however, is the need to partially formalize the requirements, albeit in a subset of natural language.
- **Automation degree & abstraction gap of generated test cases:** Another challenge is the automation degree that can be achieved. Many methods require manual interaction of a user, e.g., for the creation of a domain-specific dictionary. A trade-off between generalizability and automation degree can be observed in existing literature (refer to Chapter 3): approaches that operate on a less constrained input domain usually require more manual intervention to resolve ambiguity or to deal

with incomplete domain knowledge.

Furthermore, generated test cases are not always executable. Depending on the abstraction level of the requirements, but also on the employed approach, the manual effort to translate abstract test cases into executable ones can be significant.

- **Test suite strength & size:** Generated test suites need to be strong in order to provide value. Strength in this context is defined as a test suite’s capability to detect erroneous behavior of an SuT. If a generated test suite performs worse than a manually created test suite in detecting faults, regulations and standards will continue to enforce the manual creation of test suites. On the other side of the spectrum, generated test suites might be very strong but also very large, potentially consisting of millions of test cases. Prohibitively large test suites inhibit the applicability to software development as the time and resource usage for test execution can be a limiting factor, especially for tests that involve real hardware.

The approach presented in this thesis tries to tackle the mentioned challenges. The objectives of TG-SRL are enumerated in the following:

### Objectives

- Provide a proof of concept for a test generation framework that operates directly on natural language requirements. The input domain should be as unconstrained as possible, i.e. restrictions on the input language should be kept to a minimum.
- Achieve the highest automation degree possible: avoid the manual creation of dictionaries or domain knowledge, and avoid manual steps within the automation process. The generated test cases should be executable without manual refinement.
- Optimize the strength of the test suite while accounting for test suite size: maximize the test suite’s capability to detect system behavior that deviates from the specification, and keep its size within reasonable boundaries.

## 1.2 Solution Approach & Contributions

The method for automated test generation adopted in this thesis operates on unrestricted natural language and combines machine learning with a rule-based approach. To extract syntactic and semantic information from the natural language requirements, Semantic Role Labeling (SRL) and other Natural Language Processing (NLP) techniques are applied. Based on the extracted information, a ruleset is applied to create a formal representation of the requirements in First-Order Logic (FOL). Concrete test cases are generated via a Satisfiability Modulo Theory (SMT) solver. By modifying the FOL constraints according to defined tactics, a robust test suite with strong error detection capabilities is created. The approach is evaluated with a mutant-based strength analysis. Figure 1.1 provides an overview of TG-SRL.

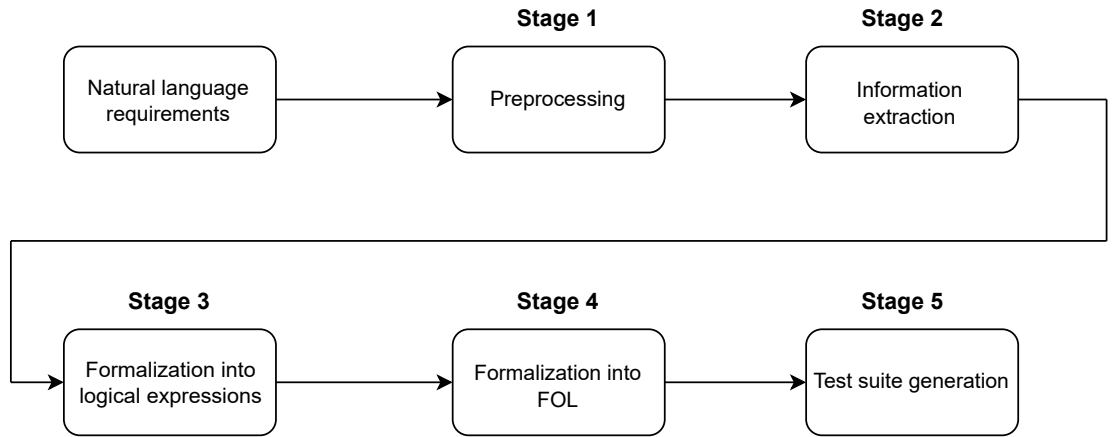


Figure 1.1: Overview of the TG-SRL approach presented in this thesis.

In the following, the main contributions of this thesis are summarized:

### Contributions

- Development of a method for extracting information relevant for test generation from natural language requirements based on the output of Semantic Role Labeling. Formalization of extracted information into logical expressions as an intermediate representation.
- Development of a method to formalize extracted information from the intermediate representation into First-Order Logic for the purpose of test generation. In particular, the method considers the need to bring the SuT into a specific state prior to test execution.
- Development of tactics for altering First-Order Logic constraints in order to generate a strong test suite.
- Analysis of existing state-of-the-art Semantic Role Labeling and Frame Semantic Parsing frameworks to extract semantic information from natural language software requirements.
- Evaluation of the presented test generation approach TG-SRL with a mutant-based strength analysis, and a comparison to *Nat2Test* from [8].
- Implementation of methods and concepts into a research framework. Publication of the framework as open source software [9].

### 1.3 Bibliographic Notes

While this thesis addresses test design and implementation, a framework for test execution and evaluation is required to assess the approach. The framework chosen in this thesis



is *Arttest* [10, 11]. *Arttest* was developed in a joint industrial project between RWTH Aachen and the Ford Research Center in Aachen. The author of this thesis together with Norbert Wiechowski and Norman Hansen were major contributors to the tool. The approach TG-SRL has not yet been published previously and has been created by the author of this thesis after finishing his work on *Arttest* independently of any industry collaboration.

## 1.4 Outline

The remainder of this thesis is structured as follows. In Chapter 2, necessary terminology and fundamentals are introduced. Chapter 3 addresses related work, while Chapter 4 presents TG-SRL in detail. In Chapter 5, TG-SRL is evaluated with a mutant-based strength analysis. Chapter 6 concludes this thesis by summarizing the results.



## 2 Fundamentals

This chapter is structured into three main sections. The first section provides an overview of the automotive product development process. The subsequent section presents an introduction to test generation from requirements. The final section introduces the fundamentals of Natural Language Processing (NLP).

### 2.1 Development of Embedded Software in the Automotive Industry

In this section, the development processes utilized in the automotive industry are introduced to establish the relevant terminology and define the scope of this thesis.

#### 2.1.1 A-SPICE & Autosar Classic

In the automotive industry, development traditionally happens according to the V-model, which breaks down product development into several phases of system and software development. During each phase, the V-model emphasizes the need for Validation & Verification (V&V) activities. Automotive SPICE (A-SPICE) [5], a domain specific variant of the international standard ISO 15504, builds on the V-model, and defines a process framework to ensure that the development process complies with defined qualitative thresholds. An overview of the A-SPICE framework is provided in Figure 2.1. A-SPICE defines several process groups that tackle different areas of product development, such as the project management process group "MAN", or the acquisition process group "ACQ". The system engineering process group "SYS" combined with the software engineering process group "SWE" represent the V-model.

**System Engineering Process Group** The development of a product starts with several system engineering activities, in which stakeholder requirements are elucidated and broken down into system and feature-level requirements. These requirements describe the behavior and architecture of the system in textual notation. In parallel, graphical notations such as UML or SysML diagrams can be used to specify the system's static and dynamic behavior. In the automotive industry, the term Model-based System Engineering (MBSE) describes a methodology that uses SysML [12] diagrams to model a system's architecture and behavior. The employed SysML diagrams heavily differ depending on the MBSE methodology, but use-case, sequence, activity diagrams, and statecharts are commonly used. MBSE offers the main benefit that the visualization of the architecture and behavioral interactions between system elements facilitates the discussion between stakeholders. In addition, complex dependencies between system elements can be better

## 2 Fundamentals

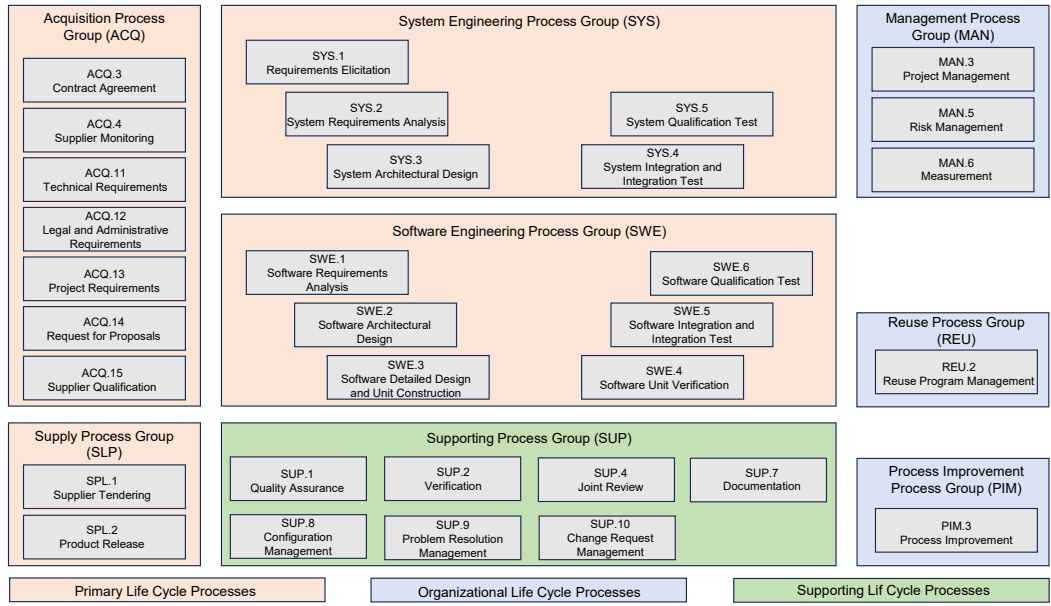


Figure 2.1: Overview of A-SPICE, adapted from [5].

captured by models in comparison to text alone.

System engineering activities can be mapped to SYS.1-3 in Figure 2.1. According to A-SPICE, it is necessary to define and plan the system integration and qualification tests for SYS.4 and 5 during these development stages.

**Software Architecture according to Autosar Classic** After the system architecture has been designed and the system has been decomposed into hardware and software, software requirements and the software architecture need to be created. A brief introduction to the currently prevailing software architecture in the automotive industry is given to establish common terminology, and clarify the scope for this thesis. The architecture described here is defined by the Autosar Classic standard [13].

Simplified, embedded software today consists of application software communicating via a communication layer with either the operating system or other application software, as depicted in Figure 2.2.

In Autosar Classic, this communication layer is called Runtime Environment (RTE). The operating system is referred to as basic software and it consists of several modules that offer services such as diagnostic event monitoring. Basic software services are heavily standardized in Autosar Classic [13]. The application software may consist of multiple levels of compositions, which in turn contain Software Components (SW-Cs). SW-Cs can contain runnables, which contain units. Figure 2.3 depicts this architecture. SW-Cs are independent of each other in Autosar Classic: an existing SW-C can be deployed on different hardware according to needs, and will communicate with other SW-Cs and the basic software via the RTE. During development of the components it is not determined

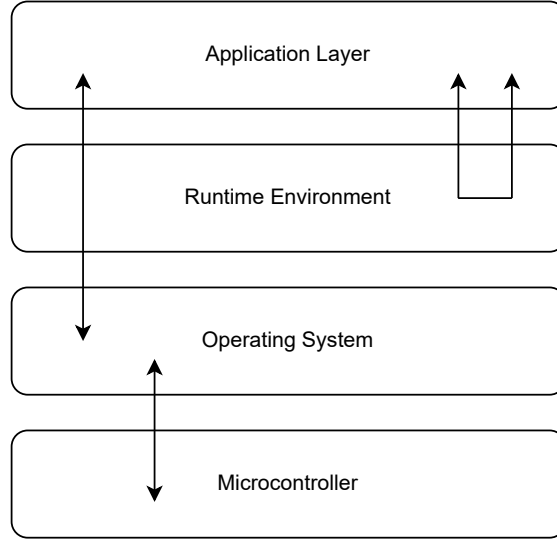


Figure 2.2: Autosar Classic architecture and communication between layers [13].

whether this communication will happen via e.g. Ethernet, CAN, LIN, or even onboard (inter-process) communication. Thus, independence of the application software from the deployment scenario is achieved. This is one major goal of Autosar Classic as it enables reuse of application software.

The requirements intended as an input to the TG-SRL approach are requirements to either application software or operating system services.

**Software Engineering Process Group** Given an Autosar Classic architecture, in SWE.1 requirements for the application software and basic software are analyzed, derived and extended. In SWE.2 and SWE.3, the design and requirements for software compositions, software components, runnables and units are derived. During the design of the software, A-SPICE foresees the creation of unit, integration, and software qualification tests which will be executed and evaluated in SWE.4, SWE.5, and SWE.6 respectively.

In the context of this thesis, as an input for the test generation, natural language requirements from either SWE.1, SWE.2, or SWE.3 are assumed. Such requirements specify application, composition, component, runnable, or unit behavior based on their input and output signals. In addition, they usually contain sufficient information to generate executable test cases. The output from the test generation approach in this thesis are thus unit or integration tests related to either SWE.4 or 5. In contrast, system requirements from SYS.1-3 are often too abstract to generate executable test cases. While there exist approaches to generate abstract test cases from system requirements, the generation of system qualification tests is not in the scope of this thesis.

Based on the requirements and architecture from SWE.1-3, the software components and units can be implemented. Implementation can be done within a Model-based Development (MBD) environment, or by directly writing code.

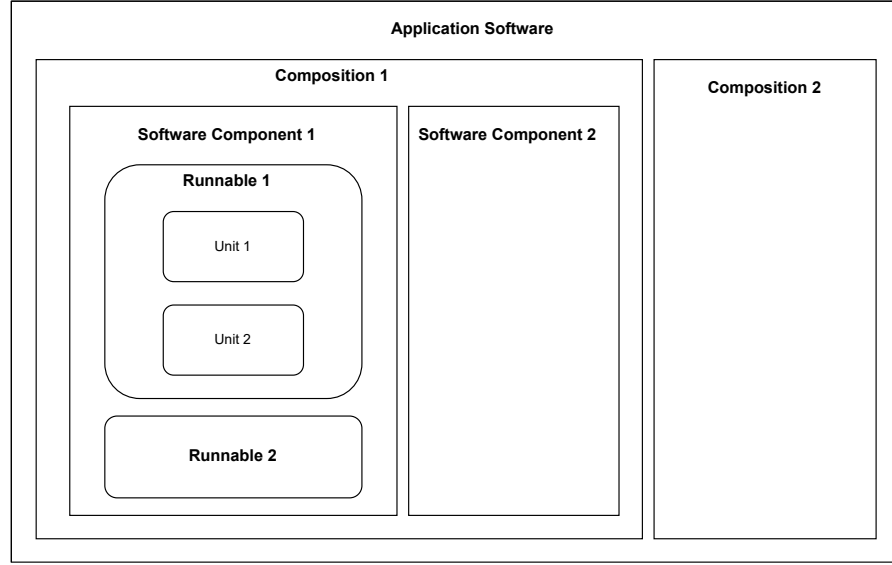


Figure 2.3: Application software structure according to Autosar Classic.

**Model-based Development** With MBD, the main development artifacts are models rather than code. Models are represented by a graphical notation with defined syntax and semantics. Based on these models, code can be generated automatically. Abstraction is a core concept of MBD. Modeling elements of a modeling language are usually domain specific and thus closer to the actual problem domain than source code of programming languages [14]. In the automotive domain, MBD enables domain experts, such as mechanical and electrical engineers, with little to no background in software development to develop automotive embedded software.

### 2.1.2 Verification & Validation Activities

Validation & Verification (V&V) activities in automotive product development encompass a variety of tasks, including static analysis, guideline checking, structural testing, and functional testing. In this thesis, the focus is on functional test cases derived from requirements.

For the scope of this thesis, test cases are defined as follows:

**Definition 2.1.1** (Test case). A set of test inputs, execution conditions, and expected results developed for a particular test objective, such as to exercise a particular program path, or to verify compliance with a specific requirement.

A test suite is defined as a collection of test cases.

Structural tests, such as coverage tests, can be generated automatically by analyzing the control and data flow of models or code, and achieving certain coverage criteria (e.g. statement, decision or branch coverage [15]). Combined with techniques from static analysis, structural test cases can, for instance, help find unreachable code or violations of

variable ranges. However, structural test cases do not check the system behavior against its specification and thus do not reveal errors in the functional behavior of code.

Typically, functional test cases are manually derived from requirements. Manual creation of such test cases has several issues. Firstly, the effort for creating test cases manually can be significant [3], thus test strategies usually consider a test end criteria to find a reasonable trade-off between test efforts and test coverage [16]. In requirements-based testing, exemplary test end criteria are requirements coverage or conformance. For example, the test strategy could specify that it is sufficient to create one test case per requirement.

A second challenge comes with changes to requirements as they necessitate an update of the derived test cases. With software changes becoming increasingly frequent in the context of SDVs, this implies growing test efforts during the post-production lifecycle of a vehicle or product.

A third challenge with both manual and automatic test generation is the quality of the requirements: natural language requirements can be ambiguous, incomplete or inconsistent. During manual specification, the tester deals with these challenges by correcting inconsistencies (if perceived), making assumptions about the behavior for incomplete requirements, and interpreting potentially ambiguous requirements. Thus, the quality of test cases is heavily dependent on the expertise and knowledge of the tester. In an automated approach, the quality of requirements themselves is critical. For instance, in the case of inconsistent requirements, a test generation framework may hint to the inconsistencies but not automatically resolve them. Chapter 4 provides details on how the proposed framework deals with ambiguity, incompleteness, or inconsistency in natural language requirements.

**Black-Box, Grey-Box & White-Box Testing** Testing methodologies can be categorized into three types: black-box, grey-box, and white-box testing [15]. If only the inputs and outputs of an SuT are accessible within a test case, this is called black-box testing. For functional testing, black-box testing is usually sufficient as requirements often specify the output behavior of an SuT based on its inputs. In contrast, structural testing necessitates white-box access to the models or code, i.e. it is necessary to be able to access internal signals and variables in order to determine the data and control flow. There are use-cases where selected internal variables need to be monitored or modified, e.g. for fault injection into a system. In case a testing framework supports monitoring and overriding selected internal signals, this is referred to as grey-box testing.

## 2.2 Test Generation from Requirements

This section introduces the fundamentals of generating test cases from requirements. Whereas Section 2.3 focuses on NLP techniques to extract information from natural language requirements, this section provides a broader overview.

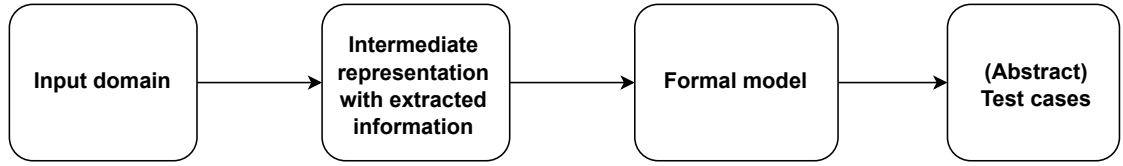


Figure 2.4: Simplified overview of model-based test generation.

### 2.2.1 Model-Based Test Generation

An overview of the general approach for model-based test generation is provided in Figure 2.4. Initially, information must be extracted from the input domain. Some approaches directly create a formal model while others utilize an intermediate representation to first collect and aggregate extracted information. Once this information is translated into a formal model, various model-specific strategies can be employed to generate test cases. Whether resulting test cases are abstract or executable depends on the approach and the input domain.

**Input Domain** The input domain can range from informal to formal notations. As such, requirements can take the form of natural language, Controlled Natural Language (CNL), use-case diagrams, or state charts that depict the system behavior. Each notation allows for an enhanced degree of formalization by imposing additional constraints on the input domain. For instance, a use-case diagram might be constrained to include only controlled natural language. Further, the use-case diagram’s structure and content may be constrained by a template. Such a template could stipulate the use of keywords or the presence of structural elements like pre- and post-conditions.

This thesis focuses on approaches that use either natural language or Controlled Natural Language (CNL) as the input domain. Approaches that rely on a fully formalized model are out of scope.

**Controlled Natural Languages** CNLs are subsets of natural language with a restricted grammar and vocabulary, designed to reduce or eliminate ambiguity and complexity. Two methods exist to derive a controlled language: either via restriction of natural language, or via constructing a CNL from the ground up by defining its grammar and syntax rules. The latter approach is used by the authors of [17] to generate test cases and is elaborated upon in Chapter 3. Examples of CNLs include Attempto [18], Simplified Technical English [19], Processable Language (PENG) [20], Gherkin [21], and SysReq-CNL [8]. Typically, a trade-off is observed between the naturalness and predictability of controlled languages [22]. Languages defined via a grammar from the ground up tend to be less natural but highly predictable. Generating tests with such languages is simpler compared to those formed by restricting natural language, which are more comprehensible and natural, albeit less predictable. For a more thorough analysis, the reader is directed to [22].



**Formalization** Information extraction is highly dependent on the formalization degree of the input domain. In the case of a CNL, information extraction can make use of the restrictions imposed by the CNL. In the case of natural language input, Natural Language Processing (NLP) techniques can help with extracting the necessary information for test generation.

Based on the extracted information, an intermediate model might be employed to aggregate and process the collected information. Following this, a formal model is created, either directly from the extracted information, or based on the intermediate representation. Examples of formal models utilized in the literature include hybrid automata, state charts, logic formula or process algebras.

One peculiarity for test generation is the encoding of time. Some methods do not account for time-dependent behavior in a test case. Those that do either use formal models that inherently handle time, such as hybrid automata, or logically encode time into the formalization. The consideration of time is vital when testing the behavior of automotive embedded systems, as a test case specifies the behavior of multiple variables or signals over a certain test duration. In the context of embedded systems, the sample time, also referred to as step size, refers to the rate at which a discrete system samples its inputs. Consequently, time can be treated as discrete, and signals can be thought of as time-value pairs as shown in Table 2.1. For a more comprehensive classification of formal models for test generation, the reader is referred to [23].

Time	Signal A
0.0	1
0.1	2
0.2	5
0.3	0

Table 2.1: Exemplary signal specified as time-value pairs. Step size of 0.1.

**Test Generation** Once a formal model is created, test cases need to be derived. The algorithm for test generation highly depends on the underlying formal model. Approaches employing automata or state charts often use structural methods to generate test cases, e.g. via path or state coverage algorithms. In contrast, if the formalization is a logical formula, a solver might be used. For process algebras, refinement checking is one valid approach to generate test cases. A test generation method from the authors of [24], which employs refinement checking on a process algebra known as Communicating Sequential Processes (CSP) [25], is detailed in Chapter 3.

The generated test cases are either abstract or can be directly executed on the SuT. Methods utilizing requirements on the system level, or less detailed inputs such as use-case diagrams, will generally only generate abstract test cases. In contrast, approaches utilizing more detailed requirements, such as software requirements, are capable of generating executable test cases.

### 2.2.2 Mutant-Based Strength Analysis

One method to evaluate the performance of a test suite is so-called mutation testing or mutation-based strength analysis. With mutation testing, the idea is to artificially modify the SuT (e.g., a program) and introduce errors. The resulting system is called a mutant. To automatically and strategically mutate an SuT, so-called mutation operators can be applied. An example for a mutation operator is replacing basic binary arithmetic operators such as  $+$  or  $*$  with other binary arithmetic operators. If the whole test suite detects the error of a mutated SuT, i.e. at least one test case of the test suite fails, the mutant is considered *killed*. If a test suite does not detect the error of a mutated system, the mutant is considered *alive*. The mutation score is calculated as the quotient of all killed mutants and the total number of mutants:

$$\text{Mutation Score} = \frac{\text{Number of Killed Mutants}}{\text{Total Number of Mutants}} \quad (2.1)$$

A higher mutation score indicates a better performing test suite.

A few limitations exist with mutant-based strength analysis. Firstly, frameworks utilizing mutation operators for automatically generating mutants often introduce one error per mutated program. While a test suite might be good at detecting single modifications of an SuT, it might not be suited to detect cases where multiple errors are introduced simultaneously. In general, mutant-based strength analysis only provides an indication for the effectiveness of a test suite to detect errors, but does not prove its effectiveness. Another disadvantage with automatic mutant generation comes with the undecidability of program equivalence [26]. Generated mutants can be semantically equivalent to the original SuT, so the mutation score is always a worst-case estimate for the performance of the test suite on the given mutants. For instance, Listing 2.1 shows several modifications that are semantically equivalent to the unmodified statement in the programming language Java. A test suite will not be able to detect such mutations, thus lowering the mutation score.

## 2.3 Natural Language Processing

The test generation method presented in this thesis operates on unrestricted natural language. In order to extract the necessary information from the natural language requirements, Natural Language Processing (NLP) techniques are employed. This section introduces relevant NLP techniques, provides a brief overview of methods for extracting semantic information from natural language, and presents popular NLP frameworks that facilitate some form of semantic information extraction.

NLP, an interdisciplinary subfield of linguistics, computer science, and artificial intelligence, aims to enable computers to understand natural language, including its contextual nuances. While NLP encompasses both written text and speech recognition, the scope of this thesis is limited to the application of NLP techniques to written text.

---

```

1  // original code
2  a = a + 1;
3  // mutant (introduce ++)
4  a = a++ + 1;
5  // mutant (introduce --)
6  a = a-- + 1;
7
8  // original code
9  if (a == true)
10 // mutant (modify condition)
11 if (a)

```

---

Listing 2.1: Code snippet with semantically equivalent mutations in Java.

### 2.3.1 NLP Methods and Common Tasks

NLP has traditionally been achieved via rule-based techniques, i.e. the hand-coding of a set of rules, also referred to as symbolic NLP [27]. In the last decades, statistical techniques [28, 29, 30, 31, 32] and large, hand-annotated training corpora [33, 34], as well as machine learning advancements have had a major impact on the field of NLP and enabled the development of sophisticated syntactical and semantic analysis [35]. While statistical techniques such as Hidden Markov Models have been very successful for certain NLP tasks, the major drawback is the reliance on elaborate feature engineering [36]. With the rise of deep neural networks and automatic / unsupervised learning methods, neural networks have become the dominant method in current NLP research [37]. Major advantages of neural networks in comparison to rule-based methods include the robustness to unfamiliar or erroneous input. In addition, a neural network’s performance can be improved by simply providing additional training data while improving a system based on handwritten rules can only be achieved by increasing the complexity of the rules, thus making maintenance and extension more difficult.

The following paragraph provides a brief overview of the most common NLP tasks, as specified by [38], which are relevant to both related work and TG-SRL. The overarching goal of TG-SRL is Natural Language Understanding (NLU), also referred to as natural language interpretation. NLU involves converting chunks of text into more formal representations, such as First-Order Logic structures, to facilitate manipulation by a computer program. NLU involves the identification of the intended semantic using an array of NLP techniques. The framework presented in this thesis primarily uses Semantic Role Labeling (SRL), which relies on several other NLP tasks, including Part of Speech (POS) tagging and Named Entity Recognition (NER).

#### Text Processing

- **Word Segmentation** This task involves separating a continuous text into individual words. As words are separated by spaces in English, this task is easy to achieve for the English language.

## Morphological Analysis

- **Lemmatization** This task involves the removal of inflectional endings and returning the base dictionary form of a word, also known as a lemma. Unlike other techniques that reduce words to their normalized form, lemmatization utilizes a dictionary to map words back to their original form. For example:

closed → close

saw → see

- **Stemming** Similarly to lemmatization, inflectional endings from a word are removed. In contrast to lemmatization, a set of rules is used instead of a dictionary.
- **Part of Speech (POS) Tagging** Given a sentence, determine the Part of Speech for each word. Some examples for POS categories include nouns, verbs, adverbs, adjectives, conjunctions, and auxiliary verbs. Words can serve as multiple parts of speech. For example, "book" can be a noun (e.g., "the book on the table") or a verb (e.g., "to book a flight"). "Set" can be a noun, a verb, or an adjective.

## Syntactic Analysis

- **Sentence Boundary Disambiguation** Given a chunk of text, find the sentence boundaries. Punctuation marks, in e.g. abbreviations, are special cases that need to be handled.
- **Parsing** This task involves generating a parse tree for a given sentence. A parse tree is an ordered, rooted tree that represents the syntactic structure of a sentence according to the language's grammar. The grammar for natural languages is ambiguous and thus, sentences can have multiple, up to thousands, parse trees. There are two primary types of parsing: dependency parsing and constituency parsing [39]. Dependency parsing focuses on the relationships between words in a sentence (marking things like primary objects and predicates), whereas constituency parsing focuses on building out the parse tree using a probabilistic context-free grammar.

## Lexical Semantics - Semantics of Individual Words in Context

- **Named Entity Recognition (NER)** Given a stream of text, determine which items in the text map to proper names, such as people or places, as well as the type of each such name (e.g., person, location, organization). Named entities can span several words.
- **Word Sense Disambiguation (WSD)** When multiple interpretations for a word are possible, the task of WSD is to select the meaning which makes the most sense in a given context. Given a specific word, a database such as WordNet [40] provides a list of associated word senses. Together with context information, WordNet is often used to infer the correct word sense.

### Relational Semantics - Semantics of Individual Sentences

- **Relationship Extraction** Given a piece of text, identify the relationships among named entities, i.e. entities identified via NER.
- **Semantic Role Labeling** Given a single sentence, identify and disambiguate semantic predicates, then identify and classify the semantic roles. Details are provided in the next subsection.

### Discourse Semantics - Semantics beyond Individual Sentences

- **Coreference Resolution** This task involves identifying words within a text that reference the same entities. An example of this is anaphora resolution, where pronouns are mapped to the corresponding nouns or names to which they refer.
- **Topic Segmentation and Recognition** Given a chunk of text, separate the text into distinct segments, each centered around a specific topic, and subsequently identify the topic for each segment.

#### 2.3.2 Semantic Role Labeling with Proposition Bank Annotations

As mentioned in the above overview, SRL identifies semantic predicates and their associated semantic roles. The following example showcases the need for a semantic analysis:

- "The system starts."
- "The system starts the database."

In this example, a syntactical analysis is not sufficient to answer the question "Which entity is started?". Figure 2.5 depicts a dependency parse tree for both sentences.

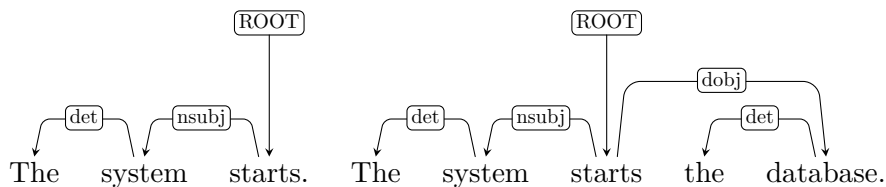


Figure 2.5: Dependency parse trees for two exemplary sentences.

In both sentences, the verb "to start" is used in active form and constitutes the root of the parse tree. "The system" is identified in both cases as the subject by a syntactical parser. However, only in the first sentence "the system" is the entity that is started. In the second sentence, the object of the sentence, namely "the database", is the entity that is being started. Thus, for the verb "to start", it depends on the semantic interpretation and not syntax of a sentence whether the object or the subject of the sentence contains the entity that is started.

SRL tries to assign specific semantic roles, also referred to as tags or thematic roles, to elements that are related to a predicate. The set of related elements together with the predicate itself are referred to as a predicate argument structure or predicate frame. A semantic role is assigned to each element in the predicate argument structure by the SRL task.

In the example above, SRL identifies "starts" as the predicate. "The system", as well as "the database", are identified as part of the predicate argument structure. Examples are depicted in Figures 2.6 and 2.7. A semantic role, in the given example "Arg1", indicates which entity is started. It is assigned to "the system" in the first sentence and to "the database" in the second sentence.

Frames for **starts** :

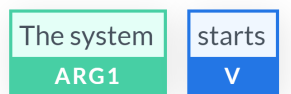


Figure 2.6: Semantic Role Labeling output from the *AllenNLP* demo [41].

Frames for **starts** :



Figure 2.7: Semantic Role Labeling output from the *AllenNLP* demo [41].

Semantic roles or tags can, for instance, indicate location, temporal attributes, or the manner in which something is done. In most SRL frameworks, semantic roles follow the ProbBank model [35], which is introduced in the following.

**Proposition Bank (ProbBank)** The ProbBank project and the corresponding model use the Penn Treebank [33] project as a base. In the Penn Treebank project, approximately 2,500 stories comprising 40,000 sentences from the Wall Street Journal were annotated with mainly syntactic information, e.g. subjects and objects of a sentence were identified. The ProbBank project added verb specific semantic annotations to the Penn Treebank corpus via a rule-based automatic tagger. The output of the tagger was manually hand-corrected [35].

Due to the difficulty of defining a universal set of semantic roles covering all types of predicates [35], ProbBank defines semantic roles on a verb-by-verb basis. The roles associated to a verb are also referred to as the verb's arguments. Arguments are numbered,

beginning with zero. In addition to the verb-specific numbered roles, ProbBank defines several general roles that can apply to any verb. These roles describe, for instance, the location or a manner in which an activity is performed. An overview of the core semantic roles defined by the ProbBank model is given in Table 2.2. A predicate together with its possible semantic roles define a so-called frameset. During the definition of the frameset, the authors of the ProbBank model tried to consistently use "Arg0" as the role that exhibits features of an agent, and "Arg1" as the role that exhibits features of a patient.

Role	Description
Arg0	Generally the argument that exhibits features of an agent
Arg1	Generally the argument that exhibits features of a patient
Arg2 & higher	No consistent generalization possible across predicates
ArgM-Adv	General adverbials or clause modifiers
ArgM-Cau	Cause
ArgM-Loc	Location
ArgM-Mnr	Manner
ArgM-Mod	Modals
ArgM-Neg	Negation
ArgM-Ext	Extent
ArgM-Dis	Discourse connectives
ArgM-Tmp	Temporal features
ArgM-Prp	Purpose or reason
ArgM-Dir	Directional features

Table 2.2: Overview of core ProbBank semantic roles.

A predicate can have multiple framesets. For instance, the predicate "to decline" can either carry the meaning of "to reject" or "to go down incrementally". The two respective framesets defined by ProbBank are shown in Figure 2.8.

**Semantic Roles of Commas** While SRL traditionally focuses on semantic roles surrounding predicates, there exists work to analyze the semantic roles surrounding commas. Commas and the surrounding sentence structure often express relations that are relevant to understanding the meaning of a sentence [42]. The following example provided by the authors of [42] illustrates this:

#### Example

We invited the computer scientists, Susan and Hannah.

Depending on the interpretation, Susan and Hannah are either a substitute for computer scientists, or the three of them form a list. The authors of [42] have defined 9 relations, shown in Table 2.3, that commas participate in.

### Frameset: decline.01 “go down incrementally”

- **Arg1:** Entity going down
- **Arg2:** Amount gone down by
- **Arg3:** Start point
- **Arg4:** End point

#### Example:

[Arg1 The net income] declined [Arg2 42%] [Arg4 to 121 million] [ArgM-Tmp in the first 9 months of 1989].

### Frameset: decline.02 “demure, reject”

- **Arg0:** Agent
- **Arg1:** Rejected thing

#### Example:

[Arg0 A spokesman] declined [Arg1 to elaborate].

Figure 2.8: Framesets from ProbBank for the word "decline".

## 2.3.3 Frame Semantic Parsing

A parallel line of work to SRL seeks to extract semantic information from sentences through a process known as Frame Semantic Parsing (FSP). FSP originates from the FrameNet project [43] and is based on a theory of meaning called frame semantics. Similar to the ProbBank project, it was the goal of the FrameNet project to annotate a specific corpus with semantic roles.

According to frame semantics, the meanings of most words can be understood on the basis of a semantic frame. A semantic frame in the context of FSP describes a type of event, relation, or entity, and the participants, also called frame elements, involved in it. As an example, the concept of cooking can be used [43]. Cooking usually involves a cook, the food that is to be cooked, a container to hold the food while cooking, and a source of heat. The notion of cooking is represented by the semantic frame "apply\_heat" in the FrameNet project. All involved participants such as the cook, food, the source of heat, and container are called frame elements. However, the semantic frame "apply\_heat" can not only be invoked by the predicate "to cook", but also verbs such as "to fry", "to bake", "to boil", or "to broil". All words that invoke a certain frame are called lexical units of the frame. An example for FSP is shown in Figure 2.9.

While in SRL with the ProbBank model, predicates act as the root of a frameset, in FrameNet also nouns, adjectives, adverbs, and prepositions are allowed as lexical units.



Relation	Description
Substitute	Indicates an apposition structure
Attribute	Separates a noun from a non-restrictive/non-essential modifier
Locative	Separates elements related to locations or places
List	Separates elements in a list
Introductory	Separates an introductory element at the start of the sentence from the main clause
Complementary	Separates a complementary element from the main clause
Interrupter	Delimits an interrupter (word, phrase or clause breaking the logical flow of a sentence)
Quotation	Separates a quote
Other	Comma that does not fall into any of the other 8 categories

Table 2.3: Semantic roles of commas according to [42].

In the depicted example, "played" invokes the semantic frame "performance\_and\_roles" with the frame elements performer, role, and performance. "Drying up" invokes the frame "becoming\_dry" with one single frame element called entity.

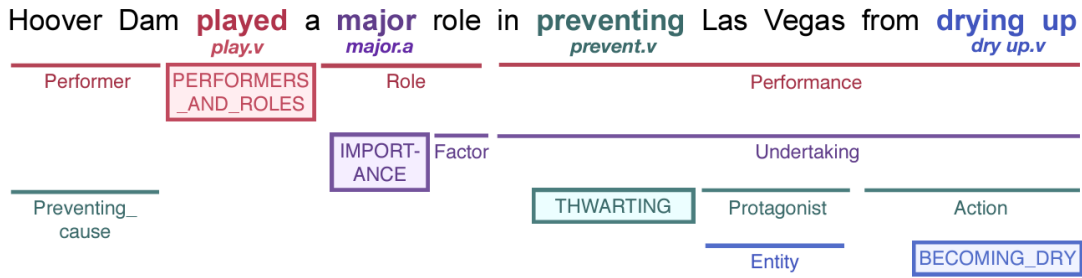


Figure 2.9: Frame Semantic Parsing example from Swayamdipta et al. [44].

In comparison to SRL, FSP and the FrameNet model emphasizes the semantics of a frame that entities are associated with. Subsequently, FSP is more domain dependent. It is essential that semantic frames have been identified and created for events, entities, or relations relevant to the domain of interest. In comparison, for SRL it is sufficient to have framesets defined for the predicates employed in the domain of interest.

In the following, two sentences annotated with ProbBank and FrameNet tags are given. These examples illustrate that ProbBank requires an additional level of inference to determine the meaning of parts of a sentence [35]. Both the lexical units "bought" and "sold" invoke the same frame "Commerce" in the FrameNet model. The buyer as well as the seller is directly identifiable based on the tags. With SRL, it is necessary to look at the different framesets of the predicates "to buy" and "to sell", and identify the buyer

and seller according to their arguments. The relevant ProbBank framesets are shown in Table 2.4.

**ProbBank:**

[Arg0 Chuck] bought [Arg1 a car] [Arg2 from Jerry] [Arg3 for 1000€]. [Arg0 Jerry]  
sold [Arg1 a car] [Arg2 to Chuck] [Arg3 for 1000€].

**FrameNet:**

[Buyer Chuck] bought [Goods a car] [Seller from Jerry] [Payment for 1000€]. [Seller  
Jerry] sold [Goods a car] [Buyer to Chuck] [Payment for 1000€].

Role	buy.01 "purchase"	sell.01 "exchange goods for money"
Arg0	Buyer	Seller
Arg1	Thing bought	Thing sold
Arg2	Seller	Buyer
Arg3	Price paid	Money earned
Arg4	Benefactive	Benefactive

Table 2.4: ProbBank Framesets for "to buy" and "to sell".

### 2.3.4 NLP Frameworks

A range of frameworks and methodologies are available publicly to tackle the challenge of NLP. For the task of SRL or FSP, most recent approaches use a layered deep neural network. Frameworks utilizing the ProbBank model for SRL include *CogComp-NLP* [45], *AllenNLP* [41], and *Sling* [46]. Frameworks using the FrameNet model include *SEMAFOR* [47] and *Open-Sesame* [44]. In general, the NLP community evaluates the performance of a neural network for NLP on certain published tasks or corpora. For SRL, one commonly evaluated dataset comes from the Conference on Computational Natural Language Learning (CoNLL) [48]. In 2004 and 2005, CoNLL provided a training corpus including correct outputs, a development data set to tune the parameters of the learning system, as well as an evaluation data set to test the performance of an SRL framework [49]. In addition to the CoNLL dataset, the OntoNotes [50] dataset is often used for training and evaluation of SRL approaches. The goal of the OntoNotes project was to annotate a large corpus comprising various genres of text (news, weblogs, talk shows etc.) in three languages, including English, with structural information and shallow semantics [51]. For the evaluation of frameworks using FSP and the FrameNet model, the data sets provided by SemEval from 2007 [52] are regularly used.

While results of frameworks for both datasets do have some carry-over to the potential performance for the purpose of extracting semantic information from natural language requirements for software in the automotive industry, the validity is still limited. All

approaches have been, to a certain degree, optimized for the published task’s datasets and thus also for the domains contained in the respective corpora. For this reason, the frameworks mentioned above have been briefly evaluated in order to make a choice for the implementation of the test generation framework TG-SRL. The evaluation can be found in Section 4.7.1 of Chapter 4.

**Semantic Role Labeling with BERT models** State-of-the-art SRL results are currently achieved with Bidirectional Encoder Representations from Transformers (BERT) [53]. BERT models are employed in both the *CogComp-NLP* and *AllenNLP* frameworks. While previous neural models for SRL relied on lexical and syntactic features such as POS tags or parse trees, BERT makes use of pre-training based on language modeling [54, 55]. The standard SRL task decomposes into four subtasks: first predicates need to be identified, afterwards the predicate sense needs to be identified (predicate sense disambiguation), next arguments for the predicate frame need to be identified and lastly, the arguments need to be classified according to the roles. For the latter three tasks, BERT together with other neural network layers can be utilized. For details of the neural network architecture, the reader is referred to [53].



## 3 Related Work

This chapter provides an overview of relevant literature in the field of test generation. Research on model-based testing techniques is very extensive as demonstrated by a survey of Dias et al. [56], which found 219 unique approaches. This chapter primarily focuses on methods that employ some form of Natural Language Processing to extract information from sentences. Methods that solely aim to formalize requirements (e.g., [57, 58]), or those that exclusively generate a formal model from requirements without producing any test cases (e.g., [59, 58]), are outside the scope of this chapter.

The methods highlighted in this chapter vary in terms of input domain, degree of automation, abstraction level of the generated test cases, and their approach to handling inconsistent, ambiguous, and incomplete requirements. The subsequent sections categorize related work according to the input domain.

### 3.1 Natural Language as the Input Domain

This section introduces relevant literature where natural language is used as the input domain.

#### 3.1.1 Solimva

Santiago et al. [60] have presented a methodology named *Solimva* to generate test cases from natural language requirements for software for space applications. The approach formalizes natural language requirements into statecharts, which are then used to generate test cases. An overview is given in Figure 3.1.

The formalization into statecharts is semi-automatic, i.e. the user needs to partially define a dictionary that represents the application domain. Within the dictionary, all properties of the statechart, such as states and transitions, are defined. Subsequently, so-called test scenarios, which describe the interaction between a user and the SuT, are manually defined. Variance in the scenarios is introduced via an automated, combinatorial approach. The natural language requirements are then manually mapped to the scenarios. Based on the scenarios, dictionary, and mapped natural language requirements, the statecharts are generated. To extract information from the requirements, POS tagging in combination with a rule-based approach is employed. Extracted information focuses on the subjects, actions, and objects of a requirement. However, a manual review of the extracted information is required as, according to the authors, the algorithm for information detection "is not general enough to work out with all natural language sentences" [60, p. 132].

Given the generated statechart, test case generation is then achieved via path traversal. Several path traversal criteria, such as transition tour and state counting, are applicable

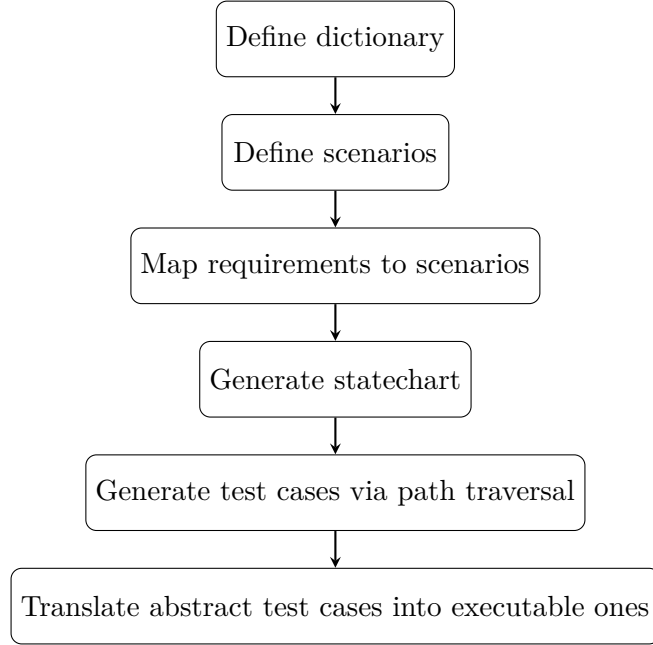


Figure 3.1: Overview of *Solimva* based on a figure in [60].

[60]. The resulting test cases are abstract and the gap towards executable test cases is closed via manual work. The approach has been compared to manually specified test cases from a domain expert and the evaluation found that the generated test cases covered more scenarios than those identified by the expert [60]. No evaluation on the capability of the generated test suite to detect faults in an SuT was performed.

Concluding, the presented approach operates on unconstrained natural language and extracts information via syntactic analysis. Several steps require manual interaction from a user and the test generation results in abstract test cases.

#### 3.1.2 Litmus

Dwarakanath et al. [6] have presented an approach named *Litmus*. A syntactic parser is employed to extract information from natural language requirements, and to identify whether a requirement is testable. A requirement is deemed testable if it contains at least one subject, an action, and optionally an object that can be identified by the parser. In addition, the identified action must contain modal verbs.

As a first step, all nouns identified by the parser are marked as entities. The set of extracted entities is then manually validated by a user. Subsequently, input sentences are broken down into less complex ones. In this phase, conditional parts of a sentence as well as conjunctions and disjunctions are split. Splitting is done via a keyword-based search on the parser output. Based on the generated simple sentences, a formalization into so-called test intents is done. Test intents are defined as "the smallest segment of a requirement sentence that conveys enough information about the purpose of the

test" [6, p. 63] and consist of a subject, an action, and optionally an object. Based on the test intents, positive and negative test cases are generated. The former verify that an action stated in the requirement does occur while the latter verify that it does not occur if the respective conditions are not met. In their work, the authors have examined abstract requirements and thus only abstract test cases are generated. The approach was evaluated on industrial requirement specification documents by manually analyzing the generated test cases for accuracy, which the authors defined as the share between correct and incorrect test cases. No evaluation on the capability of the generated test suite to detect faults in an SuT was performed.

Concluding, the presented approach operates on constrained natural language as requirements need to have certain syntactic features in order to be deemed testable. The automation degree is high as there is only user interaction required for validating the identified entities. While the generated test cases are not executable, this outcome is also attributed to the abstraction level of the input requirements. Similar to the approach *Solimva* from Santiago et al., the method extracts syntactic information in the form of subjects, actions, and objects from a sentence in order to generate test cases.

### 3.1.3 Retna

With their framework called *RETNA*, Boddu et al. [61] employ the probabilistic Charniak Parser [32] to extract syntactic information from natural language requirements. The parser tags elements of a sentence according to the Penn Treebank tagset [62]. Based on the output from the parser, the authors employ a syntax-directed semantic analysis to extract predicate-argument structures. The predicate-argument structures are then formalized into Discourse Representation Structures (DRS) [63]. An example for the formalization of a sentence into DRS is given in Figure 3.2.

#### DRS example:

"If a train is on a track then its speed should be less than the specified speed of the track."

Exists  $X1, X2$  with  $\text{isa}(X1, \text{train}), \text{isa}(X2, \text{track}), \text{ison}(X1, X2)$

$\Rightarrow$

Exists  $X3, X4, X5$  with  $\text{isa}(X3, \text{speed}), \text{of}(X1, X3),$

$\text{isa}(X4, \text{speed}), \text{of}(X4, X2), \text{isa}(X4, \text{specified}), \text{shouldbelessthan}(X3, X4)$

Figure 3.2: DRS example from [61].

Predicate structures such as " $\text{isa}(x1, \text{train})$ " from the given example are looked up in a predicate library, which contains a semantic interpretation. A user is asked to either manually verify and correct the semantic interpretation or, if necessary, manually specify a semantic interpretation. The Discourse Representation Structures are formalized into

weak monadic second order logic formula [61], which in turn are converted into state machines. Test cases are generated via path traversal of the state machine. The authors evaluated their approach in several case studies and the approach successfully discovered errors such as deadlocks in a Java program. However, in the case studies a state explosion was observed. Thus, mitigation strategies such as user steered path traversal algorithms were suggested.

Concluding, *RETNA* supports an unconstrained input domain and tackles ambiguity via a probabilistic parser. The parser extracts syntactic information, semantic information is added via a rule-based approach and via a lookup in a predicate library. Cases that are not covered by the predicate library need to be specified manually by the user. *RETNA* thus features a medium to high degree of automation. The authors did not specify whether the generated test cases were executable without additional manual effort.

## 3.2 Use Cases as the Input Domain

This section presents methods that utilize use cases as an input.

### 3.2.1 UMTS

Wang et al. [64] present an approach called *UMTS* to generate test cases based on use cases written according to the Restricted Use Case Modeling (RUCM) format [65]. The RUCM format defines a template for a use case to facilitate information extraction for the purpose of test generation. For instance, a use case according to the RUCM format contains sections about pre-conditions, post-conditions, and possible execution steps in a test case. In the original RUCM format, the content of those sections is written in unrestricted natural language. Wang et al. have extended the template with certain constraints, e.g., the sections need to contain specific keywords to enable test generation.

After formulating the requirements in use cases according to the RUCM format, the user is expected to model the domain as a Unified Modeling Language (UML) class diagram. The domain model includes all relevant entities and specifies their relationships. Based on the use cases and the domain model, pre- and post-conditions are automatically extracted via NLP methods by the framework. The employed NLP methods, such as POS tagging and tokenization, extract syntactical information from the use cases. The user then needs to manually formulate these conditions as Object Constraint Language (OCL) constraints. A constraint solver is afterwards used to solve the OCL constraints and automatically generate test cases. The authors conducted a case study where the approach was shown to cover more scenarios than a control group that wrote test cases manually [64].

Concluding, the approach operates on a significantly constrained input domain as the use case diagrams need to adhere to a template and contain certain keywords. The automation degree is low as the user is expected to formalize the extracted information into OCL, and to create the domain model as a UML class diagram. The generated test cases are executable. The capability of the generated test suite to detect faults in an SuT was not evaluated.



### 3.2.2 Text2Test

In their framework Text2Test [66], Sinha et al. employ NLP methods on use cases that can contain unconstrained natural language. The extracted information is formalized into a use case description model, which categorizes information into roles such as actor or action. For the complete model, the reader is referred to Figures 2 and 3 in [66]. The employed NLP methods are described in detail in [67].

The NLP pipeline is constructed as follows: POS tagging and shallow parsing are used to extract syntactic information. Afterwards, semantic information is added via a so-called dictionary annotator. In the dictionary, predefined semantic classes such as "to update" are available. A predicate in a given sentence is mapped to a predefined semantic class by the dictionary annotator. For instance, given a sentence that contains the verb "to change", the term is mapped to the class "to update" in the assumption that both carry the same meaning. The dictionary containing the semantic classes is domain specific and manually created. After mapping verbs to semantic classes, a context annotator classifies relevant nouns into several categories according to a rule-based scoring system. The categories distantly resemble the semantic roles from SRL. For instance, one category represents actors of a sentence. As a last step, a process builder builds the test sequence as described in the use case. Test generation then happens according to the sequence built by the process builder. Resulting test cases are abstract. The authors did not evaluate the strength of the generated test suite.

Concluding, the approach operates on uses cases that may contain unconstrained natural language. It extracts information in a predicate-centric process that is similar to SRL, and formalizes it into a use case description model. Apart from the manual creation of a domain specific dictionary, the test generation process is completely automated.

### 3.2.3 Test Case Generation, Selection and Coverage from Natural Language

In their work, Nogueira et al. [24] present a framework to generate test cases for the domain of mobile device applications. As an input, use cases that adhere to a template are used. The template specifies actions, the order of actions, the system response, and optionally conditions on the system state that need to hold true before carrying out an action. The content of the sections specified by the template needs to adhere to a Controlled Natural Language (CNL). The CNL is domain specific for the domain of mobile devices and certain actions, such as assignments to input or output signals, need to be specified in a formal notation that resembles code.

The adherence to the defined CNL is checked via a parser. For valid use cases, a formalization into Communicating Sequential Processes (CSP) [25] is done. Roughly described, CSP defines processes that communicate via events. Test generation is achieved via refinement checking as follows: a CSP formula is constructed such that the shortest counter example for the formula yields a test case. Once such a test case has been found, the original formula is extended to exclude this test case. Afterwards, the method is repeated to find the next test case. The result of the refinement checking are CSP traces consisting of events. These are mapped back to the CNL used in the use cases. In order

to achieve the goal of executable test cases, a manual translation from the CNL to an executable version is required.

The generation of a test suite can be guided by goals which need to be specified manually. For instance, a goal could either be to test whether a certain state is reachable, or to simply generate a certain amount of test cases. These constraints need to be manually encoded into the CSP formula that is used as an input for refinement checking. The authors hypothesize about possible improvements to guide the test suite generation, for instance, a transformation from the CSP notation to automata in order to employ coverage guided algorithms. In a case study, the authors evaluated the performance of the approach with regards to the time necessary for test generation. No evaluation on the test suite strength was done.

Concluding, the approach uses a significantly constrained input domain, combining templated use cases with CNLs. Given valid inputs, the formalization into CSP as well as the test generation is mostly automated. The resulting abstract test cases need to be manually translated into executable test cases.

## 3.3 Controlled Natural Language as the Input Domain

In this section, approaches that operate on CNLs are introduced.

### 3.3.1 Generating Test Cases for Timed Systems from CNL Specifications

Schnelte et al. [68] present a method to generate test cases for timed systems from CNL specifications. Listing 3.1 shows an excerpt of the grammar that defines the CNL. Input requirements need to additionally adhere to a template that defines triggers, pre- & post-conditions, and a system reaction as shown in Table 3.1. The allowed vocabulary is defined in a manually created dictionary and contains, among others, signal names and enumerations. The conditions, actions, and triggers described in the templates are formalized into temporal qualified expressions according to a ruleset. Based on the temporal qualified expressions, both positive and negative test cases are generated. The former check whether the specified behavior occurs. Negative test cases are created based on the assumption, that the requirements describe the complete behavior of the system, and that any unspecified, observable reaction from the SuT represents faulty behavior. Test generation itself happens via a reachability analysis with partial order causal link planning [69].

Concluding, the approach from Schnelte et al. operates on templated requirements that need to adhere to a language defined by a grammar. This approach significantly restricts the input domain, but removes the challenge of solving ambiguity. Given a dictionary and inputs that satisfy the constraints, the approach is highly automated, and executable test cases are generated. The strength of the generated test suite was not evaluated.

---

```

1  <preCond> := <complexCond> ( " or " <complexCond> )*
2  <complexCond> := <simpleCond> ( " and " <complexCond> )?
3  <simpleCond> := ( <forAll> | <equalsCond> ) (timeCond)?
4  <equalsCond> := <signalName> " is " <signalValue>
5  <forAll> := " all " <signalGroup> " are " <signalValue>
6  <timeCond> := <exactTC> | <forMoreTC> | <forLessTC>
7  <exactTC> := " for " <timeExpr>
8  <forMoreTC> := " for more than " <timeExpr>
9  <forLessTC> := " for less than " <timeExpr>
10 <timeExpr> := <number> <timeUnit>
11 <timeUnit> := "ms" | "s" | "m"

```

---

Listing 3.1: Grammar excerpt from [68].

Condition before trigger	Alarm is set for less than 6 s
Trigger	Driver door is open
Condition after trigger	-
Reaction	Alarm is unset

Table 3.1: Exemplary requirement, taken from [68], conforming to the grammar from Listing 3.1.

### 3.3.2 Nat2Test

In their framework *Nat2Test* [8], Carvalho et al. generate test cases based on requirements formulated in a CNL called SysReq-CNL. The SysReq-CNL was created to handle requirements from the automotive domain. Essentially, the grammar allows sentences of the following form:

When  $condition_1, condition_2, \dots, condition_N$ , the system shall:  $action_1, action_2, \dots, action_N$ .

For a formal definition of SysReq-CNL the reader is referred to Figure 3 of [8]. The grammar allows ambiguity in certain cases, and *Nat2Test* handles this by presenting possible interpretations to the user in the form of parse trees. The user then selects the parse tree that aligns with their understanding. The vocabulary that can be used in requirements is manually defined in a dictionary. The dictionary also maps each term to its lexical category such as "noun" or "verb".

As a first step in the test generation process, so-called case frames are generated by extracting thematic roles from the input requirements. These frames categorize the verbs in a sentence into two types: those present in conditions and those present in actions. In addition, each verb is associated with extracted thematic roles such as "patient" or "agent". Figure 3.3 provides an illustration of a simplified requirement and its corresponding case frame.

**Exemplary requirement:** When the voltage is greater than 80, the lights' controller component shall: assign *on* to the left indication lights.

**Case Frame:**

Condition #1 - Verb: is	
Condition Patient	the voltage
Condition From Value	-
Condition Modifier	greater than
Condition To Value	80
Action #1 - Verb: assign	
Agent	the lights' controller component
To Value	on
Patient	the left indication lights

Figure 3.3: Simplified case frame example from Table 1 in [8].

Thematic roles are extracted via a rule-based approach exploiting the syntax imposed by the grammar. The supported set of verbs is limited by rules. Action statements can contain the verbs "to add", "to assign", "to reset", and "to subtract" while conditions may contain "to be", "to become", and "to change" [8].

Based on the extracted case frames, an intermediate graph based representation called Data-Flow Reactive Systems (DFRS) [8] is generated. Given a DFRS formalization, *Nat2Test* supports the translation into three different formal models for test generation: Internal Model Representation (IMR), Communicating Sequential Processes (CSP), and Software Cost Reduction (SCR) specifications. The specific pipeline to generate executable test cases differs for all models. For detailed information on the pipeline stages, the reader is referred to [8, 70, 17].

Every formalization has its strength and weaknesses. One factor is whether the notion of time is natively supported. Moreover, computational effort due to, e.g. state explosion, poses a significant challenge for certain pipeline stages. Carvalho et al. have evaluated *Nat2Test* with the SCR and IMR formalization via a mutant-based strength analysis. The evaluation revealed that the SCR approach outperformed random testing [8], as shown in Table 3.2. The IMR approach showed comparable performance to a manually created test suite [17]. In the work "Simulation of hybrid systems from natural-language requirements" [71], Oliveira et al. extended the SysReq-CNL to support requirements that describe hybrid systems. In particular, support for mathematical functions and expressions was added.

*Nat2Test* is publicly available for download. Hence, in the evaluation presented in Chapter 5 of this thesis, TG-SRL has been compared to *Nat2Test* in a mutant-based strength analysis.

### 3.3 Controlled Natural Language as the Input Domain

System under test:	VM	NPP	PC	TIS
Java (LOC):	64	57	39	325
Mutants:	523	319	134	4320
NAT2TEST				
Killed:	302	154	128	1619
Mutation score:	57.74%	48.28%	95.52%	37.48%
Randoop				
Killed:	265	123	134	1031
Mutation score:	50.67%	38.56%	100.00%	23.87%
NAT2TEST $\times$ Randoop				
Killed only by NAT2TEST:	105	83	0	876
Killed only by Randoop:	68	52	6	288
Combined mutation score:	70.75%	64.58%	100.00%	44.14%

Table 3.2: Mutant-based strength analysis results from [8]: Nat2Test compared to random testing.

Concluding, *Nat2Test* operates on a CNL and thus on a constrained input domain. Once a user has defined a dictionary and resolved ambiguities, test case generation is fully automated, and the resulting test cases are executable.

### *3 Related Work*

## 4 Test Generation with Semantic Role Labeling

This chapter provides an in-depth explanation of the TG-SRL approach. The objectives of this framework, as detailed in Section 1.1, are to minimize constraints on the input domain, achieve a high automation degree, and to optimize the test suite for strength and size.

Several assumptions are made for the approach presented in this chapter. These were partially mentioned in Chapter 1.1 and are summarized as follows:

- The SuT is software for an embedded system. It is a software composition, component, runnable, or unit in the context of Autosar Classic (cf. 2.3), or has comparable properties.
- The software requirements are formulated in natural language. They are either requirements from the phases SWE.1, SWE.2, or SWE.3 in the A-SPICE process framework (see Section 2.1.1), or have a comparable level of detail. The requirements describe the system behavior according to concrete in- and output signals of the SuT, or at least a mapping is feasible between variables contained in the requirements and in- and outputs of the SuT.
- Certain information about signals is available a priori, e.g. in a data dictionary. In particular, it is assumed that the data type, initial value, minimum, and maximum value are known a priori.

In the next section, an overview of the test generation framework is provided. Afterwards, each test generation stage is described in detail. Lastly, implementation details, including tool specific limitations, are presented.

### 4.1 Overview

A generic pipeline to generate test cases based on natural language requirements may look as follows:

#### Generic Test Generation Pipeline

1. Collection of information from other sources than requirements to facilitate test generation.
2. Preprocessing of requirements.
3. Extraction of information via NLP or rule-based approaches.

4. Aggregation, processing, and formalization of extracted information.
5. Test generation based on formalized models via a method that highly depends on the formalization approach.

The aim of the framework proposed in this thesis is to leverage advanced NLP techniques for semantic information extraction, thereby minimizing restrictions on the input domain. As described in Section 2.3, both Frame Semantic Parsing and Semantic Role Labeling are suitable methods to achieve this goal. Ultimately, SRL has been chosen based on a brief evaluation of available SRL and FSP frameworks, which can be found in Section 4.7.1. For the formalization of requirements, a choice was made to employ First-Order Logic (FOL) formula. In the following, reasons for favoring FOL formula over alternative formalization methods, such as automata, are provided.

- The formalization from requirements into FOL formula, process algebras, or other formal languages has been used extensively in literature, as presented in Section 3. FOL is employed for knowledge representation in artificial intelligence, and is sufficiently expressive to represent natural language statements in a concise way [72, Chapter 2].
- By employing SMT solving for test generation, the gap from abstract test cases to executable test cases is bridged. Even if requirements vaguely specify variable ranges or leave certain variables unspecified, an SMT solver will assign them specific values.
- FOL formula and correspondingly SMT problem instances allow for simple modifications to create a strengthened test suite. Constraints can easily be modified, removed, or added to generate test cases to test a specific behavior of a system.
- Prior to testing a particular behavior of the SuT, it might be necessary to bring the system into a specific state. This can efficiently be achieved with SMT instances by adding constraints that describe the desired initial state.
- While the notion of time is not inherently supported with FOL formulas, time can be encoded symbolically in FOL.
- Via SMT solving, it is possible to deal with inconsistent and, to some extent, incomplete requirements. For inconsistent requirements, the SMT instance will be unsatisfiable, while for incomplete requirements, a behavior of the system will be assumed by the SMT solver.

For an approach employing SMT solving, it is essential to know initial values, data types, minimum, and maximum values for in- and outputs. These constraints are required in the SMT instance to ensure the generation of meaningful test cases. Otherwise, the SMT solver might assign random invalid values for signals to satisfy the SMT instance.

Based on the above two choices in favor of SRL and FOL formulas, the stages for the test generation pipeline presented in this thesis look as follows:



## TG-SRL Pipeline

### Stage 1

Preprocessing of requirements & collection of information from other sources.

### Stage 2

Extraction of syntactic and semantic information from requirements via NLP methods, particularly using SRL.

### Stage 3

Formalization of extracted information from Stage 2 into logical expressions according to a ruleset. The logical expressions consist of timed conditions and actions. Aggregation of information into a *system function* that describes the system behavior.

### Stage 4

Formalization of a system function into FOL formula. Use of an SMT solver to find a satisfying solution, which represents one test case.

### Stage 5

Modification of the SMT instance according to predefined tactics in order to generate a strengthened test suite.

The pipeline presented in this thesis combines machine learning in Stage 2 with a rule-based approach in Stage 3 for test generation. After extraction of syntactic and semantic information from the requirements via NLP, a formalization into logical expressions, which serve as an intermediate representation, is done. The resulting system function consists of timed conditions and actions, and aggregates all information that is relevant for test generation. Throughout the rule-based stage, several decisions are taken. For instance, ambiguity is resolved via the interpretation imposed by the applied ruleset. The system function is ultimately translated into FOL, and test cases are obtained via finding a satisfying solution via an SMT solver.

The rule-based stage developed in this thesis serves as a proof-of-concept, and rules have been implemented for selected verbs and cases only. To extend the scope of the rule-based stage of the pipeline, two strategies are possible. The first strategy involves expanding the existing ruleset to cover additional cases that may occur with natural language. The second strategy entails modifying the input requirements to ensure alignment with the existing rules, provided that such adjustments do not alter the original semantics of the requirement. An exemplary valid modification of a requirement could entail the replacement of words with synonyms that are covered by existing rules.

**Challenges** With regards to the challenges enumerated in Section 1.1, the presented pipeline behaves as follows:

**Input domain restrictions:** With the help of NLP methods such as SRL, the framework can in theory operate on unrestricted natural language requirements. Subsequently,

the proposed framework needs to deal with ambiguity, non-completeness and inconsistency of requirements. As the current state-of-the-art SRL approaches employ pre-trained neural networks (see Section 2.3.4), resulting annotations are potentially inaccurate. Thus, a manual preprocessing step is employed to ensure the extraction of appropriate syntactic and semantic information in later pipeline stages. The manual preprocessing phase will become less important with advancements in NLP and domain specific datasets to train the underlying networks. Aside from the accuracy of SRL, manual preprocessing is necessary due to the limited ruleset of Stage 3. For information which is required for test generation, but not provided by NLP methods, heuristics are utilized. These inherently limit the input domain. For cases that are not handled correctly by these heuristics, the input needs to be modified manually while retaining semantic equivalence.

**Automation degree:** The framework presented in this thesis contains two manual steps. The first step is the aforementioned manual preprocessing of the requirements. The second step is a worst-case approximation for the maximum test case duration that needs to be provided by the user and is explained in Section 4.5.1. The generated test cases are executable without any manual refinements.

**Strength & size of the generated test suite:** The size and strength of the generated test suite heavily depends on the applied tactics, which are described in detail in Section 4.6. As part of the evaluation in Chapter 5, a set of tactics has been identified that provides a good trade-off between test suite size and strength.

The subsequent sections describe the several pipeline stages in detail.

### 4.2 Stage 1: Signal Attributes & Preprocessing of Requirements

Stage 1 consists of the following steps:

- Collection of in- and output information from requirements, a data dictionary, or through other means.
- Depending on the accuracy of the SRL implementation on a given requirement: manual preprocessing of requirements such that the semantic information is extracted correctly.
- Depending on the extent of the ruleset implemented in Stage 3: apply methods to narrow down the input domain for the rule-based system (e.g., replace words with synonyms for which rules have been implemented).

**Collection of Input & Output Constraints** Signal information can be extracted either from requirements or from other sources. The collection of in- and output information from sources other than requirements depends on the SuT and other involved tools, such as a data dictionary. Generally, the aim is to collect as much information as possible

about the SuT and its signals in order to specify precise constraints for the SMT instance. The required information for the pipeline presented in this thesis includes:

- Signal type (input or output)
- Data type (number, boolean, or timer)
- Upper & lower bounds
- Initial value

The signal type is crucial for the creation of the SMT instance, as certain properties are specific to either output or input signals. Details are provided in Section 4.5. The data type, signal bounds, and initial values are added as constraints to the SMT instance to ensure the generation of valid signal specifications. For instance, for a boolean signal, a constraint is added to limit the signal values to either 0 or 1. The data type "timer" is specifically used for signals or variables representing timers. Timers have an implicit property which needs to be added as a constraint to the SMT instance: unless reset, a timer increments continuously as time in the test case progresses.

It is feasible to formulate these constraints towards signals in software requirements, and subsequently extract this information. For example, the sentence "Signal *a* is less than 100 at all times." can be interpreted as an upper value boundary for a signal *a*. However, for the prototypical implementation of this thesis, it is assumed information about signals is known a priori from external sources, such as a data dictionary, and provided to the framework as an input by the user.

**Preprocessing of Requirements** The goal of preprocessing the requirements is to either reduce the needed complexity in later pipeline stages (e.g., the rule-based stage), or to fix inaccuracies introduced by the employed NLP techniques. One preprocessing step could entail replacing words with synonyms, e.g. by looking up synonyms in a lexical database such as WordNet [40] or VerbNet [73]. This strategy can streamline the process in Stage 3 by reducing the number of necessary rules - for every group of synonymous words, only one rule would be required. Additionally, synonym replacement can enhance the results obtained from NLP. Since the neural networks employed in NLP frameworks are trained on specific corpora, replacing less common synonyms with more prevalent ones can increase the prediction accuracy of the network.

In the pipeline presented in this thesis, the preprocessing stage mainly entails the manual reordering of parts of a requirement, replacing verbs with synonyms, or placing commas in specific places to improve the accuracy of SRL. Examples are given in 4.7.2.

## 4.3 Stage 2: Information Extraction via NLP

This section presents how, and which, information is extracted from the software requirements via NLP. Stage 2 consists of the following steps:

- Creation of so-called requirement and predicate frames that serve as a source of data for all remaining pipeline stages.
- Execution of SRL per requirement. The resulting predicate argument structures are stored per predicate in the respective predicate frame.
- Identification of conditional parts of a requirement with an algorithm that employs SRL for commas.
- Identification of a verb's tense based on POS tags.

Before explaining these steps in detail in Section 4.3.1, additional assumptions and clarifications are presented in the following:

It is assumed that the requirements specify the behavior of embedded software by outlining changes in input signals and specifying the expected modifications to output signals. In particular, input signals can be modified by the user of the SuT and requirements do not specify the expected behavior of input signals. In contrast, output signals cannot be directly modified by the user of the SuT, but requirements specify the expected behavior of output signals. For TG-SRL, the following assumptions are made towards the requirements:

- Signals are directly referenced in the requirements or a mapping exists between signal names used in the natural language requirements and the signals of the concrete SuT. In the latter case, it is expected that the test automation framework substitutes the names according to the given mapping.
- Requirements are expected to be valid when taken out of context of other requirements, i.e. a single sentence is expected to contain valid information. This limitation is required as SRL operates on semantics of individual sentences. No NLP methods from the domain of discourse semantics have been employed in TG-SRL. In particular, no coreference resolution (see Section 2.3.1) is applied, thus every sentence needs to refer to relevant entities directly. Connections across multiple requirements are only made in later stages such as Stage 3. Future work discussed in Section 5.4 has the potential to render this assumption obsolete.
- It is assumed that all software requirements can be categorized into either conditional sentences or non-conditional sentences. The former contain one or several conditions and subsequent actions if these conditions are met. The latter only contain actions and it is assumed that these actions hold true under any condition.

The last assumption potentially restricts the input domain, but it aligns with related work that aims to generate test cases. The frameworks outlined in Chapter 3 in Sections 3.1.2, 3.2.1, 3.2.3, 3.3.1, and 3.3.2, all distinguish between some form of conditions or triggers and actions in requirements. The following sentences illustrate a conditional and a non-conditional requirement:

**Conditional requirement:** "If signal  $a$  changes to true, then signal  $b$  is set to 1."

**Non-conditional requirement:** "Signal  $a$  is true."

Conditional sentences describe the system's behavior via implications. Often conditional sentences adhere to the following pattern:

"If a user makes changes to certain input signals, then certain changes in the system's observable outputs or state are expected."

In this generic example, the conditional section of the sentence refers to input signals. The non-conditional section of the requirement refers to both output signals and internal variables, such as a system's state. However, software requirements might also refer to outputs and internal variables in their conditional sections:

"If a user makes changes to certain input signals given a specific system state, and certain output signals have specific values, then certain changes in the system's observable outputs or state are expected."

In such instances, a tester executing a test case needs to bring the system into the desired state exclusively by altering input variables. The same applies to automated test generation. Details about how this is accomplished using SMT solving are provided in Section 4.5.

#### 4.3.1 Requirement & Predicate Frames

For the generation of test cases, structures called *requirement and predicate frames* are created and populated with information extracted from requirements. They aggregate and store all information relevant for the remaining pipeline stages: the frames are populated in Stages 2 and 3 and their data is used across Stages 2 to 5.

Tables 4.1 and 4.2 detail the content of both requirement and predicate frames. Some content within predicate frames is only relevant for specific predicates. For instance, the field "source values" is only used for the predicate "to switch", as demonstrated in the sentence "to switch from value  $x$  to value  $y$ ", where " $x$ " is interpreted as the "source value".

Requirement Frame
List of predicate frames
Condition spans
List of conjunction start spans
List of disjunction start spans

Table 4.1: Content of a requirement frame.

---

**Predicate Frame**


---

Predicate argument structures from SRL  
 Referenced signals  
 Flag indicating affiliation to a condition  
 Tense  
 Source values  
 Target values  
 Delay and duration  
 Negation  
 Comparison operator ( $>$ ,  $>=$ ,  $<=$ ,  $<$ ,  $=$ )  
 Mathematical operator ( $+$ ,  $-$ ,  $*$ ,  $:$ )

---

Table 4.2: Content of a predicate frame.

**Semantic Role Labeling of Verbs** SRL is performed on the (preprocessed) requirements. SRL analyses all predicates in a given sentence and creates the corresponding predicate argument structures. The particular SRL pipeline varies based on the implementation. Following an evaluation of available frameworks (refer to Section 4.7.1), *CogComp-NLP* was selected for the implementation of TG-SRL. In the framework *CogComp-NLP*, the SRL pipeline utilizes Part of Speech tagging, lemmatization, dependency parsing, and Named Entity Recognition analysis. For details, the reader is referred to [45].

In software requirements for embedded systems, predicates refer to one or multiple signals and describe signal changes. Thus, it is necessary to extract the signals and their value changes from appropriate tags identified by SRL. Consider the following example and the corresponding SRL output in Figure 4.1.

"If the mode is 1 and the request timer is lower or equal to 30 then the mode should be modified to 3 and the request timer should be reset."

SRL identifies two predicates in the conditional part of the sentence and five in the non-conditional part. However, three predicates in the non-conditional section do not have any semantic roles associated to them. In such cases, no predicate frame is created. For the other cases, the identified semantic roles are stored in corresponding predicate frames. Stage 2 only stores information directly obtained from SRL, the extraction of relevant information such as the referenced signals is done in Stage 3 and is presented in detail in Section 4.4. Given the SRL output for the predicate "modified" in Figure 4.1b, the resulting predicate frame is presented in Table 4.3 (empty fields are omitted).

**Identification of Conditional Parts of a Requirement** For the creation of logical expressions and, in particular, implications in Stage 3, it is necessary to identify the conditional part of a requirement. In natural language, conditional sections of a sentence are not easily identifiable via only a rule-based approach. Table 4.4 illustrates this by showing possible rephrasings of a conditional section.

Frames for **is** :

If 

the mode
ARG1

is
V

1
ARG2

 and the request timer is lower or equal to 30 then the mode should be modified to 3 and the request timer should be reset .

Frames for **is** :

If the mode is 1 and 

the request timer
ARG1

is
V

lower or equal to 30
ARG2

 then the mode should be modified to 3 and the request timer should be reset .

(a) SRL frames from the conditional part of the sentence.

Frames for **should** :

If the mode is 1 and the request timer is lower or equal to 30 then the mode 

should
V

 be modified to 3 and the request timer should be reset .

Frames for **be** :

If the mode is 1 and the request timer is lower or equal to 30 then the mode should 

be
V

 modified to 3 and the request timer should be reset .

Frames for **modified** :

If the mode is 1 and the request timer is lower or equal to 30 then 

the mode
ARG1

should
ARGM-MOD

 be 

modified
V

to 3
ARG4

 and the request timer should be reset .

Frames for **should** :

If the mode is 1 and the request timer is lower or equal to 30 then the mode should be modified to 3 and the request timer 

should
V

 be reset .

Frames for **be** :

If the mode is 1 and the request timer is lower or equal to 30 then the mode should be modified to 3 and 

the request timer
ARG1

should
ARGM-MOD

be
V

reset
ARG2

 .

(b) SRL frames from the non-conditional part of the sentence.

Figure 4.1: Exemplary SRL tags created with the *AllenNLP* online demo [41].

Predicate Frame: to modify
Predicate argument structures from SRL: "the mode" (Arg1), "should" (ArgM-Mod), "to 3" (Arg4)

Table 4.3: Exemplary predicate frame based on SRL output in Figure 4.1.

Rephrasings of "If condition $c$ holds true, then execute action $a$ ."
When condition $c$ holds true, [...]
While condition $c$ holds true, [...]
As long as condition $c$ holds true, [...]
Execute action $a$ if condition $c$ holds true.
Given condition $c$ , [...]
Under the assumption that condition $c$ , [...]

Table 4.4: Possible rephrasings for the given sentence.

Thus, a combination of NLP methods and a rule-based approach is employed to extract the conditional parts of a requirement. TG-SRL applies the heuristic shown in Algorithm 1 to identify conditional sections within a requirement. The concept behind this algorithm is as follows: in some natural language requirements, a comma separates the conditional part of the sentence from the main clause which contains the actions. In these cases, SRL for commas [42] tags the comma as "introductory" and the conditional part of the sentence as the "introductionary part", as explained in Section 2.3.2. The following example illustrates this.

"<sup>[Introductionary Part</sup> If condition  $c$  holds true] <sup>[Introductory,]</sup> then execute action  $a$ ."

As a plausibility check, the conditional part is examined for keywords such as "if" or "when". In case SRL for commas cannot identify an introductionary part, or the sentence simply does not contain a comma, a keyword search is applied. It is assumed that the beginning of the conditional section is marked by keywords such as "if", "given", or "when". Similarly, it is assumed that the main clause starts with keywords such as "then" if no comma separates it from the conditional section. In case the keyword search is unsuccessful, it is assumed that the statements made in the requirement hold true unconditionally.

This approach to extract conditional parts of a requirement is limited for natural language. A more sophisticated key word search or other NLP methods may improve the identification of conditional parts of a sentence. For the scope of this thesis, for every requirement where the heuristic does not yield the correct results, manual modifications as part of the preprocessing activities are required. For instance, a comma may be added to a requirement such that Algorithm 1 delivers plausible results.



---

**Algorithm 1** Algorithm to identify conditional sections within a requirement.

---

**Input:** requirement**Output:** identified conditional parts**Method:**

```

if requirement contains commas then
    apply SRL for commas
    extract conditional parts based on the identified tags
    check results for plausibility
    if results are plausible then
        return identified conditional parts
    end if
end if
apply a rule-based keyword search
if the keyword search was unsuccessful then
    declare the requirement as a statement without conditional parts
end if
return identified conditional parts

```

---

Given the exemplary sentence "If signal  $a$  changes to true, then signal  $b$  is set to 1.", the heuristic identifies "If signal  $a$  changes to true," as the conditional part via SRL for commas. This information is saved in the requirement frame with the help of so-called start spans. These are defined as follows:

**Definition 4.3.1** (Start & End Span). The start/end span refers to the index of the first/last character of the referenced segment or word in the original text. It marks the start/end point of the segment or word.

In the given example, the start span of "if" is 0 while the start span of the comma is 28. The resulting predicate frame is shown in Table 4.5.

---

**Requirement Frame: "If signal  $a$  changes to true, then signal  $b$  is set to 1."**


---

Condition spans: [0, 28]

---

Table 4.5: Exemplary requirement frame with identified conditional parts.

**Verb Tense** A test case consists of time-value pairs per signal. As requirements may refer to signal changes that are in the past or in the future, it is necessary to determine the tense of a predicate. For the identification of the verb tense, POS tagging is employed as POS tags for verbs indicate their tense. For example, the tag "VBZ" indicates present tense while the tag "VBD" indicates past tense. However, the verb tense is insufficient to generate concrete test cases. As illustrated in the following example, the temporal semantics of a requirement are potentially ambiguous:

"If signal  $a$  was 10, then signal  $b$  is set to 20."

Assuming the current time is  $t$  with  $t \in \mathbb{N}_0$ , "signal  $a$  was 10" may refer to time  $t - x$  with  $x \in \mathbb{N}_{>0}$ ,  $x < t$ . The value of  $x$  depends on the interpretation. A concrete interpretation of such cases is given by the implemented ruleset, which is explained in Section 4.4.3.

**Conjunction & Disjunction Start Spans** For test generation, the logical connection between entities within requirements needs to be considered. As preparation for Stage 3, in which the logical connection is identified, the start spans for relevant keywords such as "and" and "or" are stored in the requirement frame (see Table 4.1). Relevant keywords are identified via POS tagging: conjunctions and disjunctions will be marked with the POS tag "CC".

## 4.4 Stage 3: Processing and Aggregation of Extracted Information into Logical Expressions

Stage 3 consists of the following steps:

- Processing & aggregation of information extracted in Stage 2. The results are stored in the existing predicate and requirement frames.
- Formalization of information within predicate and requirement frames into logical expressions that represent the conditions and actions contained in a requirement.

The first step entails extracting information about referenced signals, signal values, mathematical and comparison operators (see Table 4.2). As this information is highly predicate specific, processing and aggregation happens via a rule-based approach. After a review of software requirements for embedded systems and related work (e.g., verbs employed in the CNL of *Nat2Test* [8]), the decision was made to implement predicate-specific rules for the following verbs as a proof of concept:

- "to be"
- "to change"
- "to add"
- "to subtract"
- "to reset"
- "to multiply"
- "to divide"
- "to switch from [...] to [...]"

- "to set"

Future work could extend this predicate-specific ruleset to cover all predicates that are part of ProbBank. Alternatively, a user extensible dictionary could be implemented, similar to the approach chosen by Boddu et al. in their framework *RETNA* [61].

#### 4.4.1 Enriching Predicate Frames

The employed ruleset for enriching the predicate frames is not exhaustive. A base ruleset designed to extract generic information is applied to each predicate. For information that is more specific to a predicate, predicate-specific rules are utilized. The necessity of such an approach is illustrated based on the example from Figure 4.1:

"If the mode is 1 and the request timer is lower or equal to 30 then the mode should be modified to 3 and the request timer should be reset."

It is assumed that we want to extract the signal value. For the text segment "the mode is 1" and the corresponding predicate "to be", the semantic role "Arg2" refers to the signal value (refer to Figure 4.1a). In contrast, for the text segment "the mode should be modified to 3" and the corresponding predicate "to modify", "Arg4" refers to the signal value (refer to Figure 4.1b). This example illustrates that SRL necessitates an additional level of inference to determine the meaning of parts of sentence, in this case where the value of a signal is contained (refer to Section 2.3.3). Consequently, given a specific predicate, firstly the correct ProbBank frameset needs to be identified, and secondly, the frameset needs to be inspected to infer which tag contains the required information. The ProbBank frameset that is invoked by a predicate in a given context is available as part of the SRL output. The ProbBank model only defines one role for "to modify" and three roles for "to be" [74]. In the given example, for "to be", the role "be.01 - copula" is invoked in which "Arg1" is specified to contain the "topic" and "Arg2" is specified to contain the "comment". Based on these descriptions in ProbBank, a rule is implemented to search the semantic role "Arg2" for the signal value, assuming that the value is commonly located in the "comment" of the "be.01 - copula" ProbBank frame.

This approach is, however, not sufficient to produce robust results due to the current accuracy and quality of SRL implementations. In practice, the expected information is not always contained in the corresponding semantic role. Consequently, instead of simply checking a single semantic role, a hierarchical approach is chosen. Based on heuristics, several semantic roles are inspected for the required information. In the following paragraphs, a detailed explanation is given on how information from Stage 2 is processed, aggregated, and used to enrich the predicate frames. Tables 4.7 and 4.8 provide examples for fully populated predicate frames.

**Signal Extraction** For the extraction of signals that are referenced within predicate argument structures identified by SRL, tags are inspected according to a predicate-specific priority list. As previously motivated, a hierarchical approach is employed to increase

Category	Priority list	Exemplary regular expressions
Negation	ArgM-Mnr > ArgM-Loc	"* not *
Comparison Operator (" > ", " >= ", " < ", " <= ", " == ")	ArgM-Mnr > ArgM-Adv > ArgM-Tmp > Arg1 > Arg2	"* greater *", "* more *", "* higher *", "* > *"
Values	ArgM-Mnr > Arg1 > Arg2 > ...	" [0-9]+ "
Temporal Attributes	ArgM-Tmp > ArgM-Adv	" after * seconds", " for * seconds"

Table 4.6: Overview of inspected tags and examples for associated regular expressions.

robustness of the framework. The priority lists for each of the supported verbs start with the semantic role "Arg1", which generally exhibits features of a patient. As requirements refer to changes of signals, the signals can often be seen as the patients of the sentence.

The remaining part of the priority list is predicate-specific and based on heuristics. In the following, the priority list for the verb "to assign" is given. If no signal match is found in a currently examined role, the search continues with the next role.

#### Signal extraction priority list for "to assign"

1. "Arg1" (generic)
2. "Arg2"
3. "ArgM-Mnr"
4. "Arg0"

The extraction of other information such as negations, mathematical operators, and target values is achieved with a similar approach: a set of semantic roles is inspected according to a priority list and the content of a specific role is examined with the help of regular expressions. Table 4.6 provides an overview of the priority lists and corresponding regular expressions.

**Mathematical Operator** The following mathematical operators are supported in the prototypical implementation: "+, -, \*, :". The identification of the mathematical operator happens based on the predicate itself. A mathematical operator is assigned in case the predicate invokes the correct frameset according to the ProbBank model. For instance, ProbBank defines five different framesets for the verb "to add" [75], with only one role invoking the meaning of mathematically adding a value to another value. For the verbs "to add", "to subtract", "to divide", "to multiply", the corresponding mathematical operator

**Requirement:**

When  $x$  is greater than 2, then the system shall add 10 to  $y$ .

	Predicate Frame 1	Predicate Frame 2
Type	condition	action
Verb	to be	to add
Tense	present	present
Referenced signals	$x$	$y$
Source values	-	-
Target values	2	10
Comparison op.	>	=
Delay	-	-
Duration	-	-
Negation	false	false
Math. op.	-	+

Table 4.7: Exemplary predicate frames.

is added to the predicate frame in case the respective frameset with a mathematical meaning is invoked.

**Value Extraction** A requirement might refer to multiple value assignments for a single signal within a condition or action:

"Signal  $a$  is greater than 10 or equal to 5. Signal  $b$  is 1 and 2."

In most cases, only a disjunction between signal values is meaningful. In the given example, signal  $b$  can never be both 1 and 2 at the same time step. Thus, a disjunction as a logical connection between multiple signal values is assumed by default.

**Temporal Attributes** Sentences may also contain information about temporal attributes such as the duration or delay of a signal change. The following sentence illustrates this:

"When signal  $a$  is 3 for 10 seconds, then change signal  $b$  to 5 after 5 seconds."

The interpretation of these temporal attributes is often ambiguous. The framework assumes a specific interpretation for these cases, which is detailed in Section 4.4.3.

#### 4.4.2 Formalization into Logical Expressions

After enrichment of the predicate and requirement frames, the next step is the formalization into logical expressions. These expressions are created by analyzing the logical, temporal, and value dependencies between predicate and requirement frames.

It is assumed that the complete set of requirements describes the functional behavior of a single SuT. A logical expression containing all relevant information for test generation

**Requirement:**When  $x$  is 0 or 1, switch  $x$  to 10 after 3 seconds.

	Predicate Frame 1	Predicate Frame 2
Type	condition	action
Verb	to be	to switch
Tense	present	present
Referenced signals	$x$	$x$
Source values	-	-
Target values	{0,1}	10
Comparison op.	=	=
Delay	-	3
Duration	-	-
Negation	false	false
Math. op.	-	-

Table 4.8: Exemplary predicate frames.

from a requirement set is referred to as a *system function* within the scope of this thesis. Such a system function consists of other logical expressions that represent the conditions and actions of individual requirements. The structure of a system function is given in Listing 4.1.

**Function Entries** A system function consists of multiple function entries. Simplified, a function entry represents a requirement. All function entries are connected via a conjunction as it is expected that all requirements hold true simultaneously. A requirement can either contain only actions, or it specifies actions that are only applicable if certain conditions are met. A function entry formalizes this information into an implication consisting of an antecedent and consequent. In case of requirements with no conditional

```

1  <function> => <functionEntry> (and <functionEntry>)*
2  <functionEntry> => (<antecedent> | true) => <consequent>
3  <antecedent> | <consequent> => <boolExpr> (and <boolExpr>)*
4  <boolExpr> => <orExpr> | <andExpr> | <notExpr> | <atomicExpr>
5  <orExpr> => <boolExpr> (or <boolExpr>)*
6  <andExpr> => <boolExpr> (and <boolExpr>)*
7  <notExpr> => not <boolExpr>
8  <atomicExpr> => <signalExpr> [comparison operator] (<signalExpr>
9  [mathOperator])? <valueExpr>
10 <signalExpr> => [signal](t+[timeOffset])
    <valueExpr> => [value]
```

Listing 4.1: Structure of a system function.

parts, this is modeled via setting the antecedent to "true". The following examples illustrate this:

**Requirement:**

If  $\langle condA \rangle$  or  $\langle condB \rangle$ , then  $\langle actionC \rangle$  and  $\langle actionD \rangle$ .

**Resulting expression:**

$(\langle condA \rangle \text{ or } \langle condB \rangle) \Rightarrow (\langle actionC \rangle \text{ and } \langle actionD \rangle)$

**Requirement:**

$\langle statementA \rangle$

**Resulting expression:**

$true \Rightarrow \langle statementA \rangle$

However, function entries also serve to group requirements to a certain degree. In case two or more requirements contain exactly the same conditions, only a single function entry is created, and all actions from the respective requirements are connected via a conjunction.

**Boolean & Atomic Expressions** Antecedents and consequents consist of one or multiple boolean expressions. A boolean expression is a disjunction, conjunction, or negation of a so-called atomic expression. Atomic expressions are used to represent individual conditions and actions from a requirement, and they essentially contain the information from the enriched predicate frames. In particular, they also contain any relevant temporal information. In the following, a simplified example is provided.

**Requirement:**

If *signal a* is 2 for 3 seconds, then add 10 to *signal b* after 5 seconds.

**Atomic expression (condition):**

*signal a* = 2 for 3s

**Atomic expression (action):**

*signal b* = *signal b* + 10 after 5s

In this example, a concrete interpretation of the temporal relation between the condition and action is required. Figure 4.2 depicts possible semantic interpretations. The condition "*signal a* is 2 for 3s" can refer to the signal being 2 in the past 3 seconds, or in the next 3 seconds. The semantic of the action "add 10 to *signal b* after 5 seconds" is also impacted by the interpretation of the temporal behavior of the condition.

The concrete interpretation assumed in this framework is as follows. A detailed explanation is given in Section 4.4.3.

**Requirement:**

If *signal a* is 2 for 3 seconds, then add 10 to *signal b* after 5 seconds.

**Atomic expressions (condition):**

$signal\ a(t) = 2 \wedge signal\ a(t-1) = 2 \wedge signal\ a(t-2) = 2$

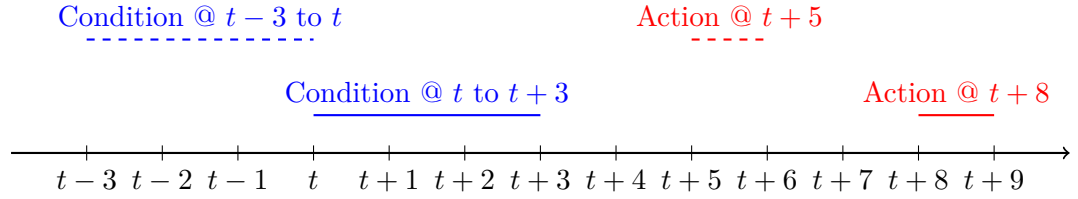


Figure 4.2: Possible interpretations of the temporal behavior for the requirement "If *signal a* is 2 for 3 seconds, then add 10 to *signal b* after 5 seconds."

**Atomic expression (action):**

$$\text{signal } b(t+5) = \text{signal } b(t) + 10$$

**Logical Connections between Atomic Expressions** Before explaining how atomic expressions for the individual conditions and actions are derived, the approach to infer the logical connection between atomic expressions is presented. However, establishing a clear logical connection is difficult due to the ambiguity of natural language. Without defined operator precedence, sentences can be interpreted in multiple ways as illustrated by the following example.

"If [...], then *signal a* and *signal b* is true or *signal c* is true and *signal d* is greater than 10 or equal to 5."

Two possible interpretations are as follows:

- "(*signal a* and *signal b* is true or *signal c* is true) and (*signal d* is greater than 10 or equal to 5)"
- "(*signal a* and *signal b* is true) or (*signal c* is true and *signal d* is greater than 10 or equal to 5)"

To resolve this ambiguity, a ruleset is implemented in which "or" takes precedence over "and". This constitutes a limitation as operator precedence in natural language requirements do not follow a strict ruleset. Additionally, the implemented ruleset only deals with a subset of logical connections within natural language sentences, namely "and", "or", and commas. In case a connection cannot be inferred, a conjunction is assumed as the default. Algorithm 2 provides an overview of the implemented ruleset and is explained in detail in the following.

For each predicate and all semantic roles of the predicate, the start and end spans are known from Stage 2. When trying to determine the connection between two conditions or actions, so-called relation spans are first calculated. Given a set of semantic roles for a predicate, the starting relation span denotes the lowest starting span across all semantic roles. The ending relation span is defined accordingly. To infer the logical connection between conditions or actions, the employed algorithm relies on the assumption that the



#### 4.4 Stage 3: Processing and Aggregation of Extracted Information into Logical Expressions

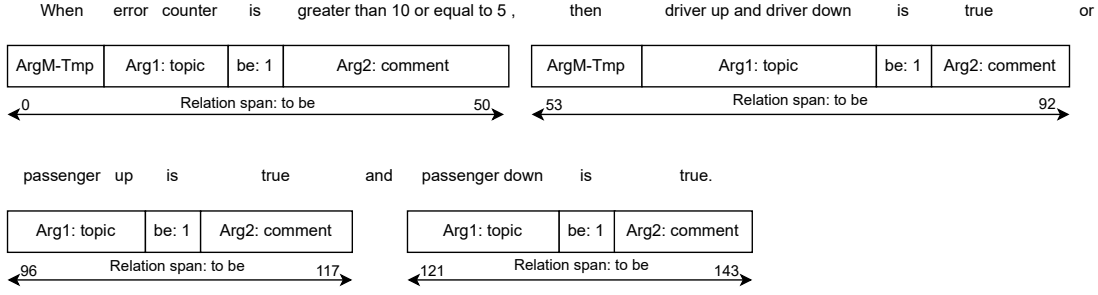


Figure 4.3: SRL output from *CogComp-NLP* [76] showcasing valid relation spans.

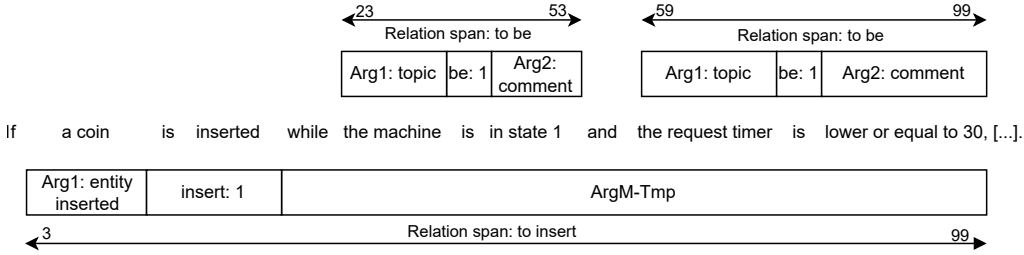


Figure 4.4: SRL output from *CogComp-NLP* [76] showcasing overlapping and thus invalid relation spans.

relation spans do not overlap. Figures 4.3 and 4.4 illustrate the relation spans and depict a valid and an invalid case, respectively. The algorithm compares the relation spans and tries to infer the logical connection by searching for keywords inbetween those spans. A conjunction is assumed as the default, e.g. in cases where relation spans overlap.

The last case described in Algorithm 2 applies if two predicate frames are separated by a comma. Such situations present a limitation due to potential discrepancies between the framework interpretation and a natural language interpretation. Consider the following example:

**Requirement:**

if  $\langle cond1 \rangle$ ,  $\langle cond2 \rangle$  or  $\langle cond3 \rangle$ , then  $\langle action1 \rangle$

**TG-SRL interpretation:**

$(\langle cond1 \rangle \text{ and } (\langle cond2 \rangle \text{ or } \langle cond3 \rangle)) \Rightarrow \langle action1 \rangle$

**Natural interpretation:**

$(\langle cond1 \rangle \text{ or } \langle cond2 \rangle \text{ or } \langle cond3 \rangle) \Rightarrow \langle action1 \rangle$

For the requirements evaluated in Chapter 5, it was verified and ensured manually that the interpretation given by the implemented ruleset matches the expected interpretation.

---

**Algorithm 2** Algorithm to infer the logical connection between predicate frames.

---

**Require:** Ordered list of predicate frames, ordered according to starting relation span

**Ensure:** List of expressions *expList* connected via a conjunction

```

1: create empty list of expressions expList for storing expressions connected via a
   conjunction
2: for each element i in the list of predicate frames do
3:   // default case for invalid frames: assume conjunction
4:   if the lowest span of frame i overlaps with the highest span of frame i − 1 then
5:     add frame i to expList
6:     continue
7:   end if
8:   if the keyword "and" is located between the lowest span of frame i and the highest
   span of frame i − 1 then
9:     add frame i to expList
10:    continue
11:  end if
12:  if an "or" is located between the lowest span of frame i and the highest span of
   frame i − 1 then
13:    remove frame i − 1 from expList
14:    add frame i − 1 to a disjunction together with frame i
15:    add the disjunction to expList
16:    continue
17:  end if
18:  // case for commas or other terms that are not supported: assume conjunction
19:  add frame i to expList
20: end for

```

---

### 4.4.3 Atomic Expressions for Conditions and Actions

In this section, expression generation for conditions and actions is presented in detail. Within Listing 4.1, the atomic expression has been specified as follows:

---

```
1  <atomicExpr> => <signalExpr> [comparison operator] (<signalExpr>
    [mathOperator])? <valueExpr>
```

---

Listing 4.2: Structure of an atomic expression.

Algorithm 3 presents how such expressions are generated. It is to be noted that one condition or action can result in multiple atomic expressions, for instance, if multiple signals or values are referenced within one condition. One atomic expression refers to a single signal and value assignment. Consequently, Algorithm 3 inspects each signal and value individually for each predicate frame.

---

**Algorithm 3** Algorithm to create atomic expressions based on a predicate frame.

---

```
1: for each signal referenced in a predicate frame do
2:   for each value assignment in the predicate frame do
3:     create an atomic expression with information from the predicate frame
4:     determine temporal semantics
5:     handle special predicates with implicit semantics
6:   end for
7:   create a disjunction between all atomic expressions resulting from multiple value
   assignments
8: end for
9: determine connection between signal specific expressions
```

---

Consider the following example to illustrate Algorithm 3:

If *signal a* and *signal b* are 10 or 20 then *<action1>*.

For each of the two signals, there are two value assignments. Thus, in total, four atomic expressions are created:

1. (*signal a*(*t*) = 10)
2. (*signal a*(*t*) = 20)
3. (*signal b*(*t*) = 10)
4. (*signal b*(*t*) = 20)

Expressions 1 and 2, as well as 3 and 4, are connected via a disjunction as described in line 7 of Algorithm 3. This rule is based on the reasoning provided in Section 4.4.1. The resulting expressions are as follows:

5.  $(\text{signal } a(t) = 10) \vee (\text{signal } a(t) = 20)$
6.  $(\text{signal } b(t) = 10) \vee (\text{signal } b(t) = 20)$

Next, expressions 5 and 6 are connected via a disjunction or conjunction according to line 9. The logical connection in such cases is inferred in a process similar to Algorithm 2. However, instead of inspecting spans of relations, spans of individual semantic roles containing the signals are inspected. The final resulting expression is as follows:

7.  $((\text{signal } a(t) = 10) \vee (\text{signal } a(t) = 20)) \wedge ((\text{signal } b(t) = 10) \vee (\text{signal } b(t) = 20))$

**Temporal Semantics** In line 4 of Algorithm 3, a concrete interpretation of temporal semantics for conditions and actions is imposed by the implemented ruleset. Consider the following requirement:

If  $x$  is 1 and  $y$  was 2, then  $\langle \text{action1} \rangle$ .

When reading such a requirement, it is natural to assume that  $x = 1$  and  $y = 2$  occur at different points in time. However, determining the exact time difference between these events is subject to interpretation. For unspecified time differences, this framework always assumes a difference of 1 step (in the context of the sample time of a system). The interpretation assumed by the framework looks as follows:

$$(x(t) = 1) \wedge (y(t - 1) = 2) \Rightarrow \langle \text{action1} \rangle \quad (4.1)$$

In addition to tenses, output variables which occur in conditions also pose a challenge in interpreting natural language requirements. Consider the following example:

**Given:**

$\text{mode}$  is an output signal

**Requirement:**

When  $\text{mode}$  is 1,  $\text{mode}$  should be set to 2.

Two possible interpretations are as follows:

$$\text{mode}(t - 1) = 1 \Rightarrow \text{mode}(t) = 2 \quad (4.2)$$

$$\text{mode}(t) = 1 \Rightarrow \text{mode}(t + 1) = 2 \quad (4.3)$$

It is unclear whether the condition refers to the current time stamp  $t$  or  $t - 1$ . To resolve this and decide on an interpretation, it is necessary to take a look at how a concrete test case is executed. Algorithm 4 illustrates the stimulation, execution, and evaluation of a single time step  $t$  for an SuT via a test engine.

As commented in Algorithm 4, the check of the output behavior happens during the currently simulated time slot  $t$ , i.e.  $\text{mode}$  is expected to be 2 at time  $t$ . Thus, when evaluating the outputs in the previous time slot  $t - 1$ ,  $\text{mode}$  must have been equal to 1.

---

**Algorithm 4** Exemplary execution of an SuT with an output variable "mode".

---

```

1: // Set inputs for time t
2: setInputs(t)
3: // mode value before execution: 1
4: executeSUT(t): if mode == 1 then mode = 2
5: // mode value after execution: 2
6: // Check outputs for time t
7: checkOutputs(t)

```

---

Subsequently, rules have been implemented to assume that a condition implicitly refers to the past in case of output variables. This corresponds to the interpretation from Equation 4.2. If a condition contains an output variable and additionally refers to it in past tense (e.g. "mode was 2"),  $t - 2$  is assumed for the time slot of the condition:

**Example:**

When the mode is 1, and the mode was not 1, assign 2 to the mode.

$$mode(t - 1) = 1 \wedge mode(t - 2) \neq 1 \Rightarrow mode(t) = 2$$

This interpretation differs from the interpretation for input variables in conditions. For those, by default, it is assumed that the input condition takes place at time  $t$  as illustrated in Equation 4.1 for the signal  $x$ .

**Duration & Delay** Apart from the above-mentioned temporal semantics, the duration and delay of a signal value change is relevant for test generation. A delay is handled by adding the delay to the calculated time slot, as in the following example:

**Requirement with a delay:**

If  $\langle condition1 \rangle$  then set  $x$  to 10 after 3 seconds.

$$\langle condition1 \rangle \Rightarrow x(t + 3) = 10$$

The duration is handled by adding a conjunction over several time slots:

**Requirement with a duration:**

If  $\langle condition1 \rangle$  then set  $x$  to 10 for 3 seconds.

$$\langle condition1 \rangle \Rightarrow x(t) = 10 \wedge x(t + 1) = 10 \wedge x(t + 2) = 10$$

The interpretation imposed by the ruleset is that the value should be held for at least 3 seconds. No assumption is made about changes prior to this duration or thereafter. In case that a duration is specified in a condition instead of an action, it is assumed that the statement refers to time slots in the past.

**Predicates with Special Rules** Some predicates have implicit semantics and are thus handled via special rules (refer to line 5 of Algorithm 3). In the following, this is illustrated for "to change" and "to reset".

For the predicate "to change", such as in the sentence "If  $x$  changes, add 1 to  $y$ ", the following expression is created:

$$x(t) \neq x(t-1) \Rightarrow y(t) = y(t-1) + 1 \quad (4.4)$$

In case the predicate "to reset" is processed, as in "If  $\langle condition1 \rangle$  then reset  $x$ ", the following expression is created:

$$\langle condition1 \rangle \Rightarrow x(t) = x_{initValue} \quad (4.5)$$

It is thus assumed that a reset assigns the initial value of the variable, if not specified otherwise (e.g. "reset  $x$  to 0").

## 4.5 Stage 4: Formalization into First-Order Logic

Based on the logical expressions inferred in Stage 3, First-Order Logic (FOL) formulae are created to represent the requirement set. The resulting set of FOL formulae is referred to as a Satisfiability Modulo Theory (SMT) instance. The SMT instance describes the behavior of the SuT as specified in the natural language requirements. A satisfying solution to the SMT instance, provided by an SMT solver, represents one test case. A test suite is constructed by modifying the SMT instance. The applied modifications are presented in detail in Section 4.6.

**Challenges in Formalization** There are several challenges in formalizing the logical expressions into an SMT instance:

1. Time dependent behavior and timers need to be encoded symbolically with existential and universal quantifiers.
2. Before being able to exercise conditions that contain output variables, it is necessary to bring the SuT into a specific state.
3. Requirements can be incomplete, ambiguous, and inconsistent. For incomplete requirements, an underspecification of output behavior is particularly challenging.
4. The satisfiability problem for FOL is, in general, undecidable [77]. Performance needs to be considered when modelling the SMT instance.

It is to be noted that some fragments of FOL are decidable. In addition, even for undecidable fragments, a decision procedure might exist that can often yield a solution. The properties of the SuT may impact in which fragment the SMT solver needs to operate. For instance, if there are only integer signals in a system, the SMT solver can apply other decision procedures or algorithms than with systems that contain both real and integer signals.

### 4.5.1 Test Case Generation

A satisfying solution of an SMT instance represents a test case. In the context of this framework, a test case starts at time step 0 and has a duration of  $d \in \mathbb{N}_0$ . A test case contains time-value pairs for each signal. For input signals, the values within the test case are treated as stimulus for the SuT, while for output signals, they represent acceptance criteria. That is, the actual output of the SuT will be compared to the specified output values in the test case.

**Algorithm for creating SMT Constraints** Given a logical expression representing a system function, as defined in Listing 4.1, Algorithm 5 is employed to create a basic SMT instance. For the test suite generation described in Section 4.6, some of these steps are modified.

---

**Algorithm 5** Algorithm to create an SMT instance for test generation.

---

**Require:** a system function  $f_{system}$  as described in Listing 4.1 which represents the requirement set.

- 1: generate constraints from  $f_{system}$
  - 2: generate min-max constraints for signals
  - 3: generate initial value constraints for signals
  - 4: generate special constraints for timers
  - 5: optionally: generate constraints for underspecified output signals
  - 6: create  $SMT_{core}$  by concatenating constraints from line 1 to 5
  - 7: enforce that one condition from a requirement is met at time  $t$ ,  $t > 0$
  - 8: create  $SMT_{ext}$  by concatenating  $SMT_{core}$  with the constraint from line 7
- 

The resulting conjunction of constraints from line 1 to 5 constitute the core of the SMT instance. Line 7 is an example for an extension of the core so that a meaningful test case is generated. By enforcing that one antecedent holds true at a certain point in time, we avoid getting a satisfying solution that does not exercise any implications formulated in the requirement set.

In line 1, the system function is traversed recursively. The resulting set of constraints represents the implications specified by the requirements. Afterwards, constraints are created to make sure signal values remain in their valid range. Line 3 serves to ensure that a test case always starts with the initial values for each signal. Listing 4.3 illustrates the constraints generated in lines 1 to 3. Due to the assumption that conditions and actions formulated in requirements are valid for the complete duration of the test case, universal quantification is utilized for these formulae. Time is encoded by specifying  $x(t)$  as a function from the integer domain to either the real, boolean, or integer domain. The latter depends on the data type of the signal. Reasons and resulting limitations for this design are given in Section 4.5.1.

The constraints formulated in line 4 from Algorithm 5 restrict valid variable assignments for timers. Without further restrictions, a satisfying solution to the SMT instance could assign arbitrary values to a timer at time  $t_1$ , given that no statements are made about

---

```

1 Requirement: "If x is 1, then increase y by 1."
2 System function and signal attributes:
3  $f_{system}: x(t) = 1 \Rightarrow y(t) = y(t-1) + 1$ 
4  $x_{min} = 0, x_{max} = 100, x_{initial} = 2$ 
5
6 SMT instance:
7  $x(0) = 2$ 
8
9 for all t:
10  $x(t) = 1 \Rightarrow y(t) = y(t-1) + 1$ 
11  $x(t) \geq 0$  and  $x(t) \leq 100$ 

```

---

Listing 4.3: Example of SMT constraints for a simple system function.

---

```

1 Given:
2 Let x be a timer variable.
3 Let C be the set of all conditions (antecedents) for which the respective
  actions (consequents) contain the variable x.
4
5 for all t:
6   if no condition in C holds true at time t:
7      $x(t) = x(t-1) + 1$ 

```

---

Listing 4.4: Example of timer constraints.

the timer behavior in the requirement set for time  $t_1$ . This poses a problem as timers are assumed to implicitly increment by 1 time unit for each time step, given that no action explicitly modifies the timer. Therefore, the constraints shown in Listing 4.4 are added for each variable of type "timer".

The constraints in line 5 from Algorithm 5 are optional and only implemented to tackle the challenge of underspecified output behavior. Figure 4.5 depicts an output signal. The signal increases linearly initially and then remains constant. A requirement set might only specify the intended behavior of such a signal for the linear section. In this case an underspecification of the signal is present. Depending on the test engine, underspecification can pose a challenge when generating test cases via an SMT solver as the SMT solver will assign random feasible values for time stamps where no condition holds true. If the test engine proceeds to compare the expected behavior of the output signal from the test case to the actual value, the test case will fail. As the test engine employed in this thesis exhibits this behavior, the constraints shown in Listing 4.5 were implemented. Some systems evaluated in Chapter 5 assume that output variables hold their value in case of underspecification, which is achieved by enabling the constraints in the aforementioned listing.



---

```

1  Given:
2  Let  $s$  be an output variable
3  Let  $C$  be the set of conditions for which the respective actions contain
   the variable  $s$ .
4
5  for all  $t$ :
6    if no condition in  $C$  holds true at time  $t$ :
7       $s(t) = s(t-1)$ 

```

---

Listing 4.5: Constraints that ensure output values do not change if no condition causes a change.

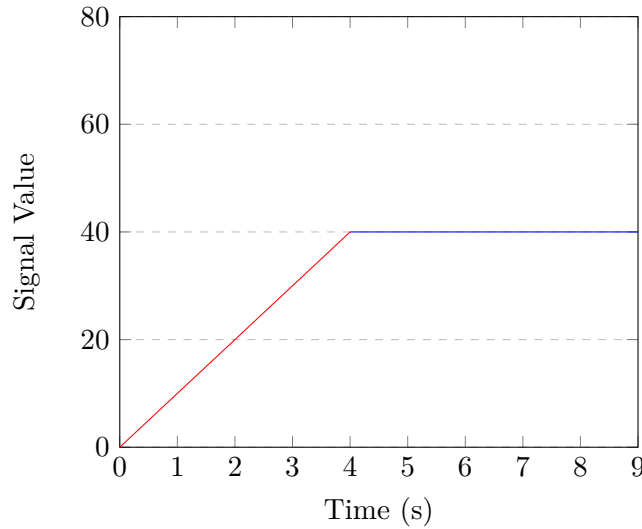


Figure 4.5: Exemplary output signal with linear and constant behavior.

**Addressing Challenges** Challenge 1 from Section 4.5 is addressed through the use of existential and universal quantifiers, as well as adding respective constraints to encode, e.g., timer behavior. Time itself has been modelled within the integer domain. Signals of data type  $dt \in \{Integer, Real, Boolean\}$  are represented as a function  $f : Int \rightarrow dt$ . This is only feasible as embedded systems operate with discrete time, and thus it can be assumed that requirements also solely refer to discrete time. In the following, two scenarios are described to first illustrate the advantage of this approach, and afterwards show limitations and necessary consequences.

**Scenario 1:** A system has a sample time of 0.1 seconds, but no requirements refer to fractions of a second. E.g., there can be requirements that specify "after 3 seconds an action happens", but there are no requirements specifying "after 0.5 seconds an action happens". In this case, during test generation, it is sufficient to only consider time steps  $t \in \mathbb{N}_0$ . This results in a significant decrease of the solution space, which becomes relevant

---

```

1   Requirement: If window position is 50, then x shall be set to 1.
2   Internal system behavior: window position initially starts at 0 and can
      only be incremented by 1 per time step
3
4   windowposition(0) = 0
5   exists  $t_1$  with  $t_1 > 0$ : windowposition( $t_1$ ) = 50
6   for all t: windowposition(t) = 50 => x(t)=1

```

---

Listing 4.6: Exemplary constraints to ensure that a certain system state is reached before executing the conditions in the antecedent.

with the performance optimizations in context of Challenge 4. These optimizations are described in detail in a subsequent paragraph.

**Scenario 2:** A system has a sample time of 0.1 seconds and several requirements refer to fractions of a second. In this case, requirements cannot be properly represented with time modelled in the integer domain without additional pre- or postprocessing steps. For instance, a factor could be applied to all time references such that all conditions and actions only refer to integers. During testing, the test engine would then need to revert this multiplication before stimulating the SuT and evaluating the output signals. These steps can theoretically be automated if the tester provides a multiplication factor as an input to the test generation framework. However, this has not been addressed in the context of this thesis as none of the evaluated system requirements in Chapter 5 referred to fractional time values.

In order to address Challenge 2 from Section 4.5, several constraints are added to the SMT instance. As explained previously, if a condition from a requirement contains output variables, the condition can only be met after bringing the system into a specific state by manipulating the input signals. However, depending on the system, reaching such a state can be non-trivial. When executing test cases, an SuT is usually in its initial state. Thus, the test case itself needs to drive the system to the state that enables the condition to be tested. Via existential quantification, constraints can be added to the SMT instance to ensure that the system state is reached at a certain point in time  $t \in \mathbb{N}_0$ . Listing 4.6 illustrates this.

Challenge 3 from Section 4.5 about incomplete, inconsistent, and ambiguous requirements is addressed as follows: in case of inconsistent requirements, the SMT instance will be unsatisfiable. An unsatisfiable core of the SMT instance can often be provided as feedback to the engineer writing the requirements. Ambiguous requirements are resolved through a concrete interpretation imposed by the rules implemented in the framework. This interpretation from the framework can deviate from the interpretation of the requirements author. In case of incomplete requirements, the SMT solver might assign arbitrary values to respective variables. This can result in failing test cases, in which case manual inspection of the test results might be necessary with the aim to add additional requirements.

---

```

1  Given:
2   $f_{system}: x(t) = 1 \Rightarrow y(t) = 2$ 
3   $x_{min} = 0, x_{max} = 100, x_{initial} = 2, d_w = 2$ 
4
5  SMT instance:
6   $x(0) = 2$ 
7
8  // for all t:
9  //  $x(t) = 1 \Rightarrow y(t) = 2$ 
10  $x(0) = 1 \Rightarrow y(0) = 2$ 
11  $x(1) = 1 \Rightarrow y(1) = 2$ 
12  $x(2) = 1 \Rightarrow y(2) = 2$ 
13
14 // for all t
15 //  $x(t) \geq 0$  and  $x(t) \leq 100$ 
16  $x(0) \geq 0$  and  $x(0) \leq 100$ 
17  $x(1) \geq 0$  and  $x(1) \leq 100$ 
18  $x(2) \geq 0$  and  $x(2) \leq 100$ 

```

---

Listing 4.7: Universal quantifier elimination example.

**Performance Optimizations** Challenge 4 from Section 4.5 relates to the performance of test case generation. For the applicability in real-world applications, the time it takes to find a satisfying solution for a given SMT instance is critical. In order to generate a complete test suite, a significant amount of SMT instances need to be solved. Initial evaluations with a concrete SMT solver implementation from Microsoft [78, 79] have shown that the performance for solving even a single SMT instance drastically deteriorates with the introduction of both existential and universal quantifiers.

As a consequence, measures were taken to eliminate the universal quantifiers to improve performance. This is achieved by introducing a worst-case bound  $d_w$  for the test case duration. Instead of specifying constraints over all possible time steps  $t \in \mathbb{N}_0$ , the constraints are simply enumerated for each time step  $t$  with  $0 \leq t \leq d_w$ . Listing 4.7 elucidates this. All universal quantifiers in the previously mentioned constraints are replaced via this approach. The worst-case bound for the test case duration is an input to the framework that is manually specified by the user, e.g. a tester. With system knowledge, it is often possible to specify a worst-case bound. The tighter this bound, the faster a given SMT instance can be solved to generate a test case. Nevertheless, specifying a bound that is too tight can impede test generation as the bound might not offer sufficient time to bring the SuT into the required states.

Another opportunity to improve the applicability in real-world scenarios is to tackle the time it takes to execute individual test cases. This can be achieved by trying to minimize the test case duration. Test cases for embedded software are either simulated with discrete time or executed on real hardware, and thus a reduction in the test case duration linearly scales with the time to execute a test case. Minimizing a certain variable

in an SMT instance, such as the variable for the test duration, can be achieved via many approaches. A naive approach is given in Algorithm 6. Many SMT solvers implement more sophisticated approaches to minimize an SMT instance according to a given objective. Details of such approaches are out of the scope of this thesis.

---

**Algorithm 6** Simple approach to minimize the duration of a test case via SMT solving.

---

```

1: initialize SMT instance  $instance_{smt}$ 
2: solve  $instance_{smt}$  and save resulting duration in  $t_{min}$ 
3: while true do
4:   add the constraint "exists  $t$  with  $0 \leq t < t_{min}$  to  $instance_{smt}$ "
5:   if  $instance_{smt}$  is unsatisfiable then
6:     return  $t_{min}$ 
7:   else
8:     solve  $instance_{smt}$  and save resulting duration in  $t_{min}$ 
9:   end if
10: end while

```

---

## 4.6 Stage 5: Test Suite Generation

The previous section presented how to generate the core SMT instance for a given system function. This section describes the modifications made to the SMT instance to generate a test suite. It is to be noted that modification possibilities are very vast and thus, in theory, exceedingly large test suites could be generated. However, test suite size does not scale linearly with a test suite's capability to detect behavior deviating from the specification (i.e. test suite strength). There are diminishing returns for adding more test cases, and the goal is to find a reasonable trade-off between test suite strength and test suite size.

As shown in Listing 4.1, every function entry represents an implication. In order to generate at least one test case that exercises the antecedent and consequently tests the implication, one SMT instance is generated for every antecedent in a function. Listing 4.8, in particular line 21, illustrates this. This strategy corresponds to the extension mentioned in line 7 in Algorithm 5. Given  $n$  function entries representing implications, this approach results in  $n$  different SMT instances and thus test cases.

Another strategy to reinforce the test suite is to test whether the system precisely implements a specified requirement. For instance, a system could implement "antecedent  $\Rightarrow$  true" instead of "antecedent  $\Rightarrow$  consequent". To discover such cases, for each implication one SMT instance with the constraint "antecedent  $\Rightarrow$  not(consequent)" is created. Resulting test cases are then expected to fail. It is to be noted that this modification of an implication needs to be applied to the SMT core as well, or else the SMT instance will be unsatisfiable. Listing 4.9 illustrates this strategy when applied to the previous example from Listing 4.8. This strategy also results in  $n$  additional test cases for  $n$  function entries if, and only if, all of these resulting instances are satisfiable.

---

```

1  Given:
2   $f_{system}: x(t) = 1 \Rightarrow y(t) = 2$ 
3   $x_{min} = 0, x_{max} = 100, x_{initial} = 2, d_w = 2$ 
4
5  SMT instance:
6   $x(0) = 2$ 
7
8  // for all t:
9  //  $x(t) = 1 \Rightarrow y(t) = 2$ 
10  $x(0) = 1 \Rightarrow y(0) = 2$ 
11  $x(1) = 1 \Rightarrow y(1) = 2$ 
12  $x(2) = 1 \Rightarrow y(2) = 2$ 
13
14 // for all t
15 //  $x(t) \geq 0$  and  $x(t) \leq 100$ 
16  $x(0) \geq 0$  and  $x(0) \leq 100$ 
17  $x(1) \geq 0$  and  $x(1) \leq 100$ 
18  $x(2) \geq 0$  and  $x(2) \leq 100$ 
19
20 // enforce antecedent
21 exists  $t_0$  with:  $x(t_0) = 1$  and  $0 < t_0 < d_w$ 

```

---

Listing 4.8: SMT instance that illustrates the enforcement of an antecedent in line 21.

---

```

1  Given:
2   $f_{system}: x(t) = 1 \Rightarrow y(t) = 2$ 
3   $x_{min} = 0, x_{max} = 100, x_{initial} = 2, d_w = 2$ 
4
5  SMT instance:
6   $x(0) = 2$ 
7
8  // for all t:
9  //  $x(t) = 1 \Rightarrow \text{not}(y(t) = 2)$ 
10  $x(0) = 1 \Rightarrow \text{not}(y(0) = 2)$ 
11  $x(1) = 1 \Rightarrow \text{not}(y(1) = 2)$ 
12  $x(2) = 1 \Rightarrow \text{not}(y(2) = 2)$ 
13
14 // for all t
15 //  $x(t) \geq 0$  and  $x(t) \leq 100$ 
16  $x(0) \geq 0$  and  $x(0) \leq 100$ 
17  $x(1) \geq 0$  and  $x(1) \leq 100$ 
18  $x(2) \geq 0$  and  $x(2) \leq 100$ 
19
20 // enforce antecedent
21 exists  $t_0$  with:  $x(t_0) = 1$  and  $0 < t_0 < d_w$ 

```

---

Listing 4.9: SMT instance that illustrates constraints for a negative test case.

**"Or" Variation Tactics** An additional strategy to strengthen the test suite is to generate variations for clauses that are part of a disjunction. Consider the following example:

**Given:**  $\langle \text{conditionA} \rangle \text{ or } \langle \text{conditionB} \rangle \Rightarrow \langle \text{actionA} \rangle$

**Testcase 1:**  $\langle \text{conditionA} \rangle \text{ is true and } \langle \text{conditionB} \rangle \text{ is false} \Rightarrow \langle \text{actionA} \rangle$

**Testcase 2:**  $\langle \text{conditionA} \rangle \text{ is false and } \langle \text{conditionB} \rangle \text{ is true} \Rightarrow \langle \text{actionA} \rangle$

**Testcase 3:**  $\langle \text{conditionA} \rangle \text{ is true and } \langle \text{conditionB} \rangle \text{ is true} \Rightarrow \langle \text{actionA} \rangle$

For  $n$  clauses within a disjunction, there are  $2^n - 1$  possible variations. Tactics can be defined to achieve certain coverage for the disjunction. Three tactics have been implemented in this framework:

### "Or" Variation Tactics

- None
- Extended (ext)
- All

For the tactic "none", no variations are generated. For the tactic "all", all  $2^n - 1$  possible variations are generated. The "extended" tactic creates expressions such that every clause is tested in isolation, and in combination with other clauses. It offers a trade-off between coverage and number of generated expressions. The "extended" tactic generates test cases according to the following strategy:

- One test case where every clause is set to true
- For each clause  $i$ , a test case where the clause  $i$  is true and the others are "don't cares"
- For each clause  $i$ , a test case where the clause  $i$  is true and the other clauses are false

For the disjunction  $x \vee y \vee z \vee u$ , Table 4.9 presents the generated expressions.

**Comparison Operator Tactics** To strengthen the test suite further, expressions containing comparison operators can be exploited. For each antecedent containing comparison operators, the original expression can be replaced with a disjunction to explicitly cover more cases. In combination with the "or" variation tactic from above, this subsequently results in multiple test cases. For the constraint " $x \geq 2 \Rightarrow y = 1$ " with  $x \in [0, 30]$ , it is possible to infer the following disjunction:

$$(x > 2 \vee x = 2 \vee x = 30) \Rightarrow y = 1$$

Two comparison operator tactics have been implemented in this framework:

	Expression
1.	$x \wedge y \wedge z \wedge u$
2.	$x$
3.	$y$
4.	$z$
5.	$u$
6.	$x \wedge \neg y \wedge \neg z \wedge \neg u$
7.	$y \wedge \neg x \wedge \neg z \wedge \neg u$
8.	$z \wedge \neg x \wedge \neg y \wedge \neg u$
9.	$u \wedge \neg x \wedge \neg y \wedge \neg z$

Table 4.9: Expressions produced by the "extended" tactic for the disjunction  $x \vee y \vee z \vee u$ .

Comparison operator	Expression
$x = y$	$x = y$
$x > y$	$x > y \vee x = x_{max}$
$x \geq y$	$x > y \vee x = y \vee x = x_{max}$
$x \leq y$	$x < y \vee x = y \vee x = x_{min}$
$x < y$	$x < y \vee x = x_{min}$

Table 4.10: Overview of disjunctions generated with the "normal" comparison operator tactic.

### Comparison Operator Tactics

- None
- Normal

The tactic "normal" produces a disjunction of clauses that depends on the comparison operator. Table 4.10 provides an overview.

Both the "or" variation and comparison operator tactics can be configured individually for test cases that are expected to fail, or those that are expected to pass. Concluding, the algorithm implemented in TG-SRL to generate a test suite is depicted in Algorithm 7.

## 4.7 Implementation

This section presents frameworks and tools employed for the implementation of TG-SRL. Additionally, it provides insights into the required preprocessing activities.

---

**Algorithm 7** Algorithm to generate a test suite with TG-SRL.

---

**Require:** system function  $f$

```

1: initialize empty test suite  $t$ 
2: generate SMT core  $c$  as described in Algorithm 5
3: // generate test cases that should pass
4: for each antecedent  $a$  in  $f$  do
5:   if comparison operator tactic is set to normal then
6:     replace relevant constraints in  $a$  with a disjunction as shown in Table 4.10
7:   end if
8:   create an empty list of expressions  $l$ , add  $a$  to  $l$ 
9:   if "or" variation tactic is set to extended or all then
10:    split all disjunctions in  $a$  as described in Section 4.6
11:    add all resulting expressions to  $l$  and remove  $a$  from  $l$ 
12:  end if
13:  for each expression  $i$  in  $l$  do
14:    enforce that  $i$  is true at a certain point in time  $t_1$ 
15:    conjunct  $c$  and  $i$  and solve the SMT instance
16:    add the resulting satisfying solution as a test case to  $t$ 
17:  end for
18: end for
19: // generate test cases that should fail
20: for each implication  $a \Rightarrow c$  in  $f$  do
21:   modify the SMT core  $c$  such that  $a \Rightarrow \text{not}(c)$ 
22:   // continue as with the passing test cases, see line 5 - 17
23:   [...]
24: end for
```

---



### 4.7.1 Frameworks and Tools

For the implementation of the pipeline presented in Section 4.1, an NLP framework and an SMT solver is required. For the latter, the Microsoft *z3* SMT solver is used [78]. Several of the built-in features are utilized in Stage 4 and 5, for instance:

- Optimizing a given SMT instance with regards to the value of a term  $t$ . This is used to minimize the test case duration when generating a test case.
- The tactic "split-clause" which, for a clause  $Or(f_1, \dots, f_n)$  splits it into  $n$  subformulas  $f_i$ . This tactic is employed when generating the "or" variations.
- Support of quantifiers. Existential quantifiers are used in the SMT instances as explained in Section 4.5.
- Extraction of an unsatisfiable core. The unsatisfiable core is useful to identify which parts of the requirement set are inconsistent (e.g., for debugging or providing feedback to a requirements engineer).

### NLP Framework

As mentioned in Section 2.3.4, a range of frameworks exist to tackle the challenge of NLP, and, in particular, Semantic Role Labeling (SRL) or Frame Semantic Parsing (FSP). A choice for the NLP framework *CogComp-NLP* was made based on a brief evaluation of available frameworks, which is presented in this section. Criteria that were considered for the evaluation are as follows:

- Availability of an online demo
- Availability as a pre-compiled library or package for Windows
- Regularly updated: contributions have been made in the last 2 years
- Availability of NLP methods: SRL or FSP, POS Tagging

For the evaluation, the following exemplary sentence has been used.

"If a coin is inserted while the machine is in state 1 and the request timer is lower or equal to 30, then the machine state should be modified to 3 and the request timer should be reset."

The results presented in the following have been created on 31.03.2023.

**CogComp-NLP** The output of the semantic role labeler in the *CogComp-NLP* framework [76] is given in Figure 4.6. Notably, the *CogComp-NLP* framework also implements an extension of SRL for nouns (SRL-Nom) [80] and prepositions (SRL-Prep) [81]. *CogComp-NLP* is distributed as a Java Maven package.

**AllenNLP** The output of SRL from *AllenNLP* [41] is shown in Figure 4.7. It is very similar to the output given by *CogComp-NLP*, with one exception: for the predicate "to modify", *CogComp-NLP* identifies two additional roles, namely "ArgM-Adv" and "ArgM-Tmp". *AllenNLP* is distributed as a Python package.

**Sling** *Sling* [46] is distributed as a Python package for Linux, and does not offer an online demo. A brief evaluation on Ubuntu showed similar output to that of *CogComp-NLP* and *AllenNLP*. The output can be found in the appendix in Table 7.6.

**Open-Sesame** *Open-Sesame* [44], a framework utilizing FSP, does not have an online demo and is not available for Windows. In addition, at the time of writing this thesis, the last contributions were made 3 years ago. Thus, the framework was not considered for the implementation of TG-SRL.

**SEMAFOR** *SEMAFOR* [47] also utilizes FSP. However, the framework is no longer maintained, and the authors refer to *Open-Sesame* on their GitHub [82]. In addition, the online demo is not functional at the time of writing this thesis. Thus, the framework was not considered for the implementation of this thesis.

Concluding, only two frameworks met a majority of the evaluation criteria: *AllenNLP* and *CogComp-NLP*. *Sling* produced comparable output, but is neither runnable on Windows nor is an online demo available. Ultimately, the choice was made for *CogComp-NLP* for the following reasons: it is distributed as a Java package, thus enabling easy integration into the source code of TG-SRL. Furthermore, it offers extensions of SRL, e.g. SRL for commas, nouns and prepositions. Additionally, *AllenNLP* is in maintenance mode while *CogComp-NLP* is still actively maintained as of 30.03.2023.

#### 4.7.2 Preprocessing of Requirements

The required manual preprocessing activities highly depend on the employed NLP framework. During the creation of this thesis, *CogComp-NLP* modified its SRL pipeline to utilize BERT. With this modification, a significant improvement in the accuracy and quality of the SRL output has been observed, thus reducing the need of preprocessing.

Nevertheless, certain manual preprocessing steps are still obligatory, partly due to the accuracy of SRL and partly due to limitations of the implementation of this framework. With regards to the former, Figure 4.8 illustrates a case where placing a comma differently severely impacts the roles identified by the SRL framework. In the following, several categories are enumerated where preprocessing is required.

- **Replacing enumerations with integer values**

*Reason:* Enumeration to integer mapping has not been implemented as part of this research framework, but is in theory possible, especially if enumerations are clearly identifiable (e.g., via a priori knowledge from a data dictionary).

*Example:* "If the machine is in idle state, do <action>."

*Replacement:* "If the machine is in state 1, do <action>."

- **Replacement of signal names**

*Reason:* Mismatches in SRL task.

*Example:* "If the flashing timer is greater than 200, then <action>."

Flashing is falsely recognized as a predicate.

*Replacement:* "If the timer is greater than 200, then <action>."

- **Reordering or rewriting sentences such that conditions, actions, conjunctions, and disjunctions are properly identified by the framework**

*Reason:* Conditions, actions, conjunctions, and disjunctions are all evaluated based on a ruleset. The ruleset is limited and also enforces a semantic interpretation that might not match the expectation of a human reader. Operator precedence is one example where manual preprocessing might become necessary.

*Example:* In general,  $(a \text{ and } b) \text{ or } (c \text{ and } d)$  cannot be interpreted by the currently implemented ruleset, and a remodeling to  $\neg[(\neg a \text{ or } \neg b) \text{ and } (\neg c \text{ or } \neg d)]$  is necessary.

- **Resolving inconsistent, incomplete, or ambiguous requirements**

*Reason:* Inconsistent requirements impede test generation as the SMT solver will always return unsatisfiable. Incomplete requirements might lead to meaningless test cases because the system behavior is not specified precisely enough. In case of ambiguous requirements, it needs to be ensured that the interpretation of the framework matches the expectation.

- **Replacement of verbs with verbs supported by TG-SRL**

*Reason:* This prototypical framework only implements rules for a small set of verbs, which are enumerated in Section 4.4.

The research framework TG-SRL is implemented in Java and made available as open source [9]. The repository contains the complete source code and all files related to the evaluation in Chapter 5, such as requirements, system implementations, test drivers, and generated test cases.

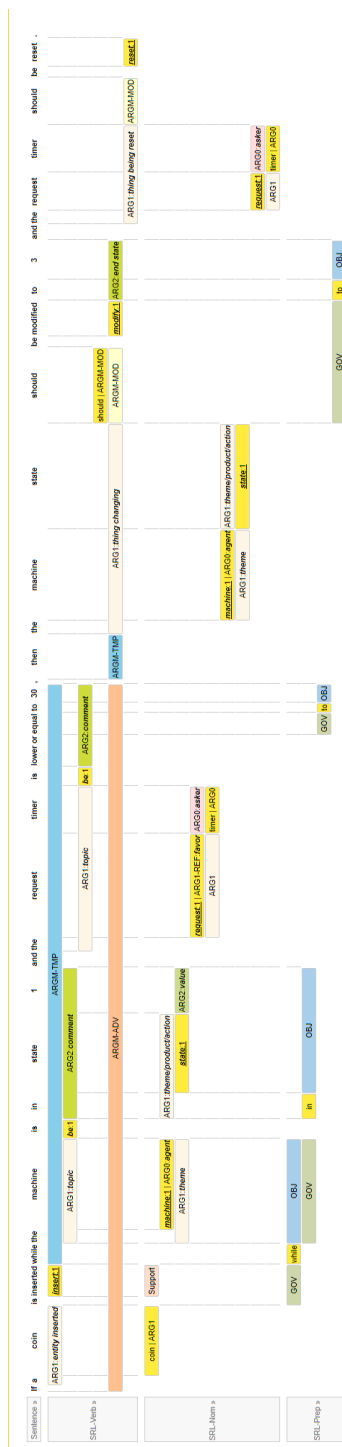


Figure 4.6: Exemplary output from *CogComp-NLP* [76].

Frames for **is** :

If a coin **is** inserted while the machine is in state 1 and the request timer is lower or equal to 30 , then the machine state should be modified to 3 and the request timer should be reset .

Frames for **inserted** :

If a coin **is** inserted while the machine is in state 1 and the request timer is lower or equal to 30 , then the machine state should be modified to 3 and the request timer should be reset .

Frames for **is** :

If a coin is inserted while the machine **is** in state 1 and the request timer is lower or equal to 30 , then the machine state should be modified to 3 and the request timer should be reset .

Frames for **is** :

If a coin is inserted while the machine is in state 1 and the request timer **is** lower or equal to 30 , then the machine state should be modified to 3 and the request timer should be reset .

Frames for **should** :

If a coin is inserted while the machine is in state 1 and the request timer is lower or equal to 30 , then the machine state **should** be modified to 3 and the request timer should be reset .

Frames for **be** :

If a coin is inserted while the machine is in state 1 and the request timer is lower or equal to 30 , then the machine state should **be** modified to 3 and the request timer should be reset .

Frames for **modified** :

If a coin is inserted while the machine is in state 1 and the request timer is lower or equal to 30 , then the machine state **should** be modified to 3 and the request timer should be reset .

Frames for **should** :

If a coin is inserted while the machine is in state 1 and the request timer is lower or equal to 30 , then the machine state should be modified to 3 and the request timer **should** be reset .

Frames for **be** :

If a coin is inserted while the machine is in state 1 and the request timer is lower or equal to 30 , then the machine state should be modified to 3 and the request timer **should** be reset .

Figure 4.7: Exemplary output from *AllenNLP* [41].

Frames for **be** :

If the system mode is 1 and the request timer is lower or equal to 30 then the system should be modified to 3 and **the request timer** should **be** **reset** .  
ARG1 V ARG2

Frames for **be** :

If the system mode is 1 and the request timer is lower or equal to 30 then the system should be modified to 3 , and the request timer should **be** reset .  
V

Figure 4.8: A comma placed before the "and" results in erroneous output. Example from *AllenNLP* [41].

## 5 Evaluation

For the evaluation of the presented approach "Test Generation with Semantic Role Labeling (TG-SRL)", a mutant-based strength analysis is done. The subsequent three systems specified by natural language requirements have been evaluated:

### Window Control

A self-designed system which is implemented as a *Matlab/Simulink* [83] model.

### Vending Machine

An example evaluated in the work from Carvalho et al. [8]. The system is implemented in Java.

### Turn Indicator System

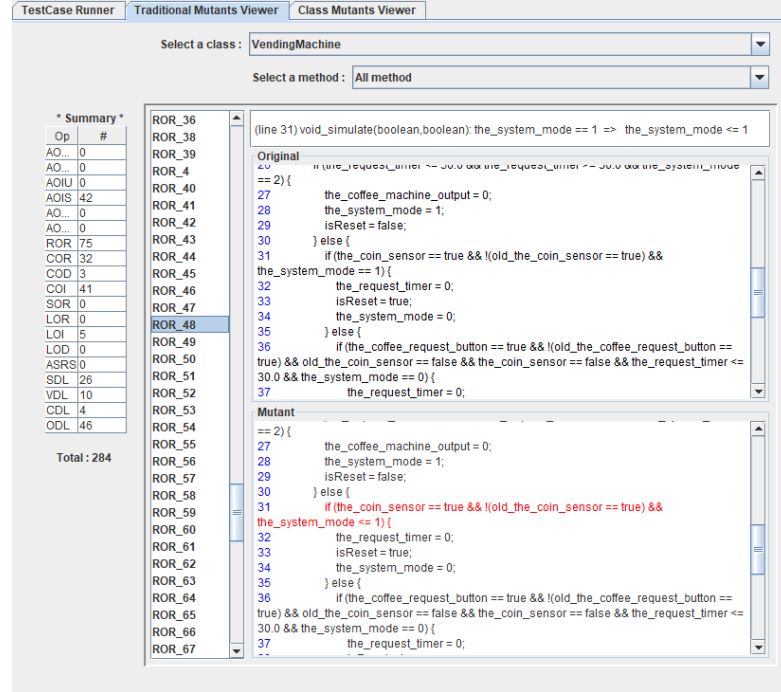
An example evaluated in the work from Carvalho et al. [8]. The system is implemented in Java.

*Matlab/Simulink* is a tool for functional behavior modeling in the automotive industry and provides simulation and code generation capabilities. For the system implemented in *Matlab/Simulink*, mutations have been created manually. For the systems implemented in Java, mutations have been generated using *MuJava* [84]. *MuJava* is a mutation system that can generate both method-level and class-level mutations by using mutation operators [85]. Figure 5.1 illustrates how *MuJava* presents method-level mutations of a program to the user. In the given figure, line 31 is mutated.

As mentioned in Section 2.2.2, tools such as *MuJava* can generate mutants that are semantically equivalent to the original system. Thus, where practical, a manual analysis has been done to ensure that any alive mutants are not equivalent. As a configuration for *MuJava*, all method-level operators are used while no class-level mutations are generated. It is to be noted that mutations generated via *MuJava* only contain a single introduced deviation.

Where feasible, the achieved mutation score of TG-SRL is compared to the mutation score of the test suite generated by *Nat2Test* from Carvalho et al. [8], which is available as a download from [86]. The authors have implemented and published multiple formalization methods as described in Section 3.3.2, however, test generation in the publicly available tool is only supported via Communicating Sequential Processes (CSP). Thus, for the remainder of this chapter, the CSP formalization is referred to when talking about test generation in *Nat2Test*.

All relevant artifacts such as *Matlab/Simulink* models, test cases, and system implementations have been published in a public Git repository [9]. The system that was used for the evaluation has the following specification:

Figure 5.1: *MuJava* method-level mutation example.

- OS: Windows 10
- CPU: i7-6700K @ 4GHz
- GPU: NVidia GTX 1080
- RAM: 32 GB
- SSD: Samsung 950

## 5.1 Window Control System

The *Window Control* system is a self-designed system for controlling a window in a car. The natural language requirements specifying the system's behavior are also self-written. A black-box view can be seen in Figure 5.2, a more detailed view of the underlying implementation is given in Figure 5.3.

The system has six input variables and five output variables, which are specified in Table 5.1. The complete natural language requirement set specifying the system consists of 32 requirements and is given in the appendix in Table 7.1.

A particularity of this system is the behavior of "current\_position": the output signal "current\_position" indicates the window position and can only be incremented or decremented by 1 each time step until it reaches either the minimum or maximum value of 0 or 100, respectively. This poses a challenge as illustrated with the following requirement:



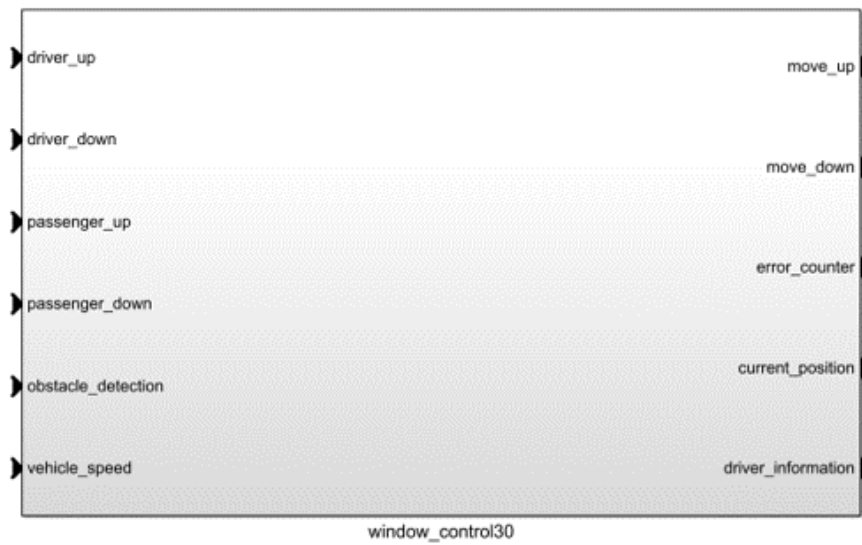


Figure 5.2: Black-box view on the *Window Control* system.

- 24: "When *move\_up* is true and *current\_position* is 100, then *current\_position* is 100."

A test case needs to bring the system into a state where the "current\_position" is 100 in order to test requirement 24. As the initial value of the signal is 0, it takes at least 100 time steps to reach this state.

During the evaluation of the system, it became evident that the requirements need to be formulated in a precise manner. Consider the following example:

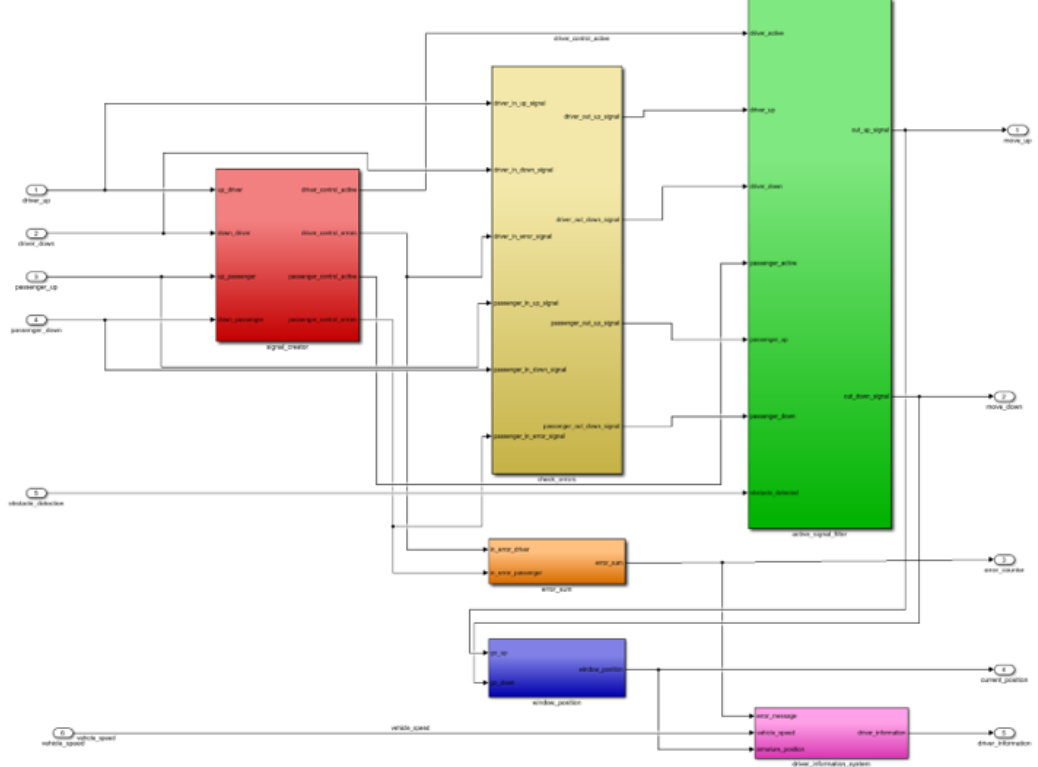
"When *passenger\_up* and *passenger\_down* is true, then the system shall add 1 to *error\_counter*."

With this specification, it is valid to increase the "error\_counter" by 1 for each time step at which both passenger signals are true. If instead "error\_counter" shall only be increased once when both signals become true, the requirement needs to be rephrased as follows:

"When *passenger\_up* and *passenger\_down* changes to true, then the system shall add 1 to *error\_counter*."

### 5.1.1 Mutant-Based Strength Analysis

30 mutations of the original *Window Control* system have been created manually and are thus guaranteed to have different behavior. A complete list with the applied mutations and expected deviation from the default behavior can be found in the appendix in Table 7.2. A *Matlab/Simulink* feature called variant subsystems [87] has been used to enable

Figure 5.3: Detailed view on the *Window Control* system.

the execution of all mutated variants for a single test case. Figure 5.4 shows the variant subsystems of the *Window Control* system. Given a test suite of size  $n$ ,  $n * 30$  simulations were necessary to calculate the mutation score. The generated test cases in the form of ".csv" files were imported into the testing tool *Arttest* [10, 11] and *Arttest* was used to both execute and evaluate the test suite in *Matlab/Simulink*.

The mutation testing results from the test suite generated by TG-SRL are presented in Table 5.2. Two configurations were used for the evaluation, which differ in the applied "or" variation and comparison operator tactics (refer to Section 4.6). The configuration is given as a tuple, with the first entry indicating the configuration for test cases that are expected to pass, and the second entry indicating the configuration for test cases that are expected to fail.

As a worst-case upper bound for the test duration  $d_w$ , a value of 150 was assumed. This estimate is due to the fact that reaching the maximum value of the "window\_position" can take 100 steps. 50% overhead was then added for potential changes after reaching a certain system state. For instance, after reaching the window position, some additional steps might be necessary to exercise another condition at time point  $t$  with  $100 \leq t \leq 150$ .

Without applying any of the tactics, the mutation score is 93.33%. Mutation 29 is left alive, which causes the following deviation from the default behavior: if the passenger presses the buttons to close and open the window simultaneously, and the driver presses

Signal	Type	Datatype	[Initial Value, Lower Bound, Upper Bound]
driver_up	input	bool	[0,0,1]
driver_down	input	bool	[0,0,1]
passenger_up	input	bool	[0,0,1]
passenger_down	input	bool	[0,0,1]
obstacle_detection	input	bool	[0,0,1]
vehicle_speed	input	number	[0,0,300]
move_up	output	bool	[0,0,1]
move_down	output	bool	[0,0,1]
error_counter	output	number	[0,0,2]
current_position	output	number	[0,0,100]
driver_information	output	number	[0,0,2]

Table 5.1: Input and output signals from the *Window Control* system.

either the button to close or to open the window, the system will not set the error counter to 1 and the window will continue closing or opening. The intended behavior is that the error counter is set to 1 and the window stops at its current position. The following requirements should cover this behavior:

- 4: "When *passenger\_up* and *passenger\_down* is true and *driver\_up* or *driver\_down* is false, then the system shall set *error\_counter* to 1."
- 18: "When *error\_counter* is greater than 0, then *move\_down* is false and *move\_up* is false."

When analyzing the test suite, it was evident that only a test case was generated where all conditions are met (i.e., "passenger\_up" and "passenger\_down" are true, "driver\_up" and "driver\_down" are false). In order to trigger the erroneous behavior, a test case is required that sets "passenger\_up" and "passenger\_down" to "true" while setting only one of the signals "driver\_up" or "driver\_down" to "true", and the other to "false". This case is addressed via the "or" variation tactic and thus the mutant has been successfully killed with the respective configuration.

The execution and evaluation time in Simulink scales linearly with the test suite size. In contrast, the more sophisticated variation tactics had an exponential impact on the test generation time, although only approximately 50% more test cases have been generated. This is because the number of SMT instances does not correlate directly to the number of test cases. When generating both comparison operator and "or" variations, many SMT instances are unsatisfiable and thus do not result in a test case.

**Nat2Test** *Nat2Test* requires input sentences that adhere to the SysReq-CNL. As summarized in Section 3.3.2, the grammar allows sentences of the following structure:



Figure 5.4: Variant subsystems for the *Window Control* system to facilitate mutation testing.

	<b>TG-SRL</b> or tactic: (none, none) comp. tactic: (none, none) $d_w$ : 150	<b>TG-SRL</b> or tactic: (ext, ext) comp. tactic: (normal, normal) $d_w$ : 150
Requirements	32	32
Mutations	30	30
Test cases	62 (43 pass, 19 fail)	92 (69 pass, 23 fail)
Time to generate tests	<10s	459s
Number of simulations	1922	2760
Test execution time in	5h 23m	8h 20m
Simulink		
Mutants killed	29/30	30/30
Mutation score	93.33%	100%

Table 5.2: Mutation testing results for the *Window Control* system.

When  $condition_1, condition_2, \dots, condition_N$ , the system shall:  $action_1, action_2, \dots, action_N$ .

Unfortunately, the CNL is too restrictive and some requirements from the *Window Control* system cannot be modelled with *Nat2Test*. The two main limitations are as follows:

- It is not possible to model a disjunction within the action part of a sentence. All actions are put into a conjunction. The following requirement cannot be modelled in SysReq-CNL.

"When  $error\_counter$  is 0, then  $driver\_up$  or  $driver\_down$  is false and  $passenger\_up$  or  $passenger\_down$  is false."

- While the verb "to change" is supported, the implementation does not support variables as values. The following requirement cannot be modelled in SysReq-CNL. Multiple different formulations are provided, however, neither are supported.

"When *current\_position* changes its value, then *move\_up* is true, or *move\_down* is true."

"When *current\_position* changes from *current\_position* to *current\_position* + 1, then *move\_up* is true. When *current\_position* changes from *current\_position* to *current\_position* - 1, then *move\_down* is true."

"When *current\_position* increments by 1, then *move\_up* is true. When *current\_position* decrements by 1, then *move\_down* is true."

Nevertheless, by omitting certain parts of the requirements, a successful CSP generation was achieved. However, test generation was unsuccessful as it did not terminate after running for more than 72 hours. A probable explanation is a state explosion as the window position has a relatively big value range of 0 to 100. Due to the above reasons an evaluation of the *Nat2Test* framework with the *Window Control* system was infeasible.

## 5.2 Vending Machine System

The *Vending Machine* system is an example with temporal properties taken from the *Nat2Test* website [86].

The system is described by five requirements in SysReq-CNL, which are given in the appendix in Table 7.4. It can also be characterized by a state machine as shown in Figure 5.5. Each requirement essentially encodes a transition between the system modes.

In Table 5.3 the input and output signals are specified. The enumerations of the signals "system mode" and "coffee machine output" are mapped to integers as shown in Table 5.5. The manually preprocessed requirements, that have been used as in input for TG-SRL, are available in the appendix in Table 7.3. Notable changes include the replacement of enumerations with integer values, and modifying the syntax of each requirement so that the output of SRL is accurate.

The *Vending Machine* system has two particularities:

1. Requirements refer to a request timer. The request timer is expected to increment by 1 in each time step if it is not reset in that same time step.
2. The requirements are incomplete. In particular, the output signal behavior is underspecified. For instance, the requirement set does not specify the behavior of the signal "coffee machine output" in case the system is in another state than "preparing strong coffee" or "preparing weak coffee". Implicitly it is assumed that the output does not change outside these two cases.

Due to the property specified in 2, the SMT instances were generated with the aforementioned optional constraints (see Line 5 in Algorithm 5) for the evaluation of the *Vending Machine*.

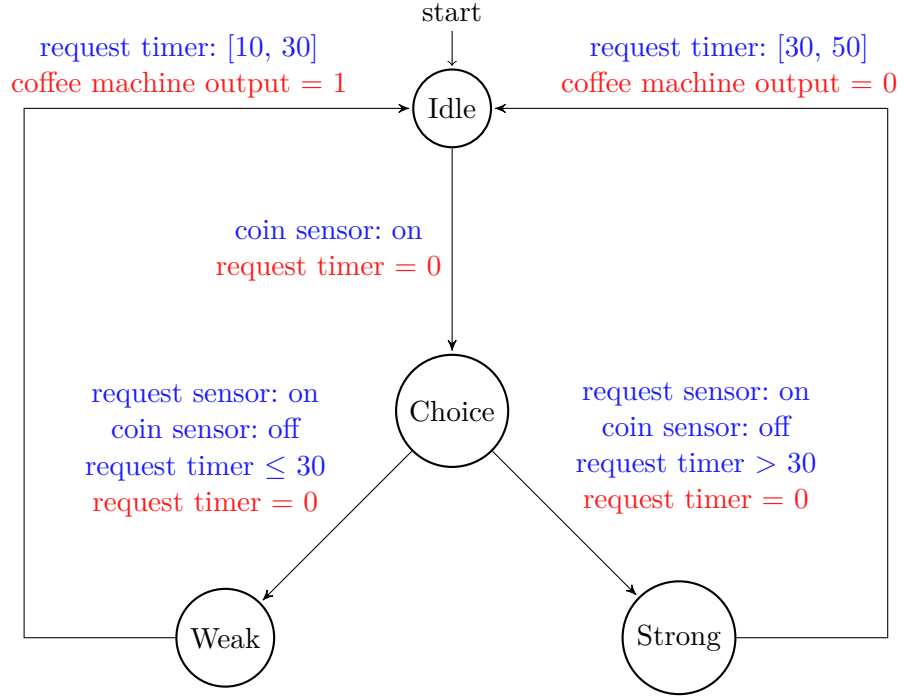


Figure 5.5: State machine for the *Vending Machine* system. Conditions are colored in blue while actions are colored in red.

In Table 5.6, a test case generated via *Nat2Test* is shown. Table 5.7 illustrates an excerpt of a test case generated via TG-SRL, the full test case is provided in Table 7.7 in the appendix. As a worst-case upper bound for the test duration a value of 150 is assumed. This estimate is made because certain conditions refer to a timer value of 50. After a reset of the timer, it takes at least 50 steps to reach this state. 100 steps have been added as an overhead to provide the SMT solver with enough space to create more complex test cases if required. There are three major differences between the test cases of both frameworks. These stem from conceptual differences with regards to how time is encoded and handled in both frameworks:

1. The test cases from TG-SRL contain the request timer as an explicit signal.
2. Time in *Nat2Test* test cases is dense, while in the test cases from TG-SRL it is discrete.
3. The average number of signal changes of a test case is significantly lower in the *Nat2Test* test cases.

Difference 1 stems from the fact that *Nat2Test* treats timers as internal variables during test generation. The expectation is that the SuT properly handles the request timer and resets it. By excluding the timer from the generated test cases, it is not possible to check the correct behavior of the timer, i.e. whether it is reset under correct conditions.

Signal	Type	Datatype	[Initial Lower Bound, Upper Bound]
the coin sensor	input	bool	[0,0,1]
the coffee request sensor	input	bool	[0,0,1]
the request timer	output	timer	[0,0,60]
the system mode	output	number	[1,0,3]
the coffee machine output	output	number	[0,0,1]

Table 5.3: Input and output signals from the *Vending Machine* system.

Signal	Mapping
the system mode	Idle: 1 Choice: 0 Strong: 2 Weak: 3
the coffee machine output	Strong: 0 Weak: 1

Table 5.4: Enumeration to integer mapping for the *Vending Machine* system.

The property described in 2 results from how time is encoded in both approaches. *Nat2Test* encodes time as reals and thus employs a dense time domain, while the framework presented in this thesis encodes time as integers and thus employs a discrete time domain. The reasoning behind modeling time as integers is given in Section 4.5.1.

Property 3 is a consequence of how time is handled during test generation. *Nat2Test* only enumerates time steps in a test case at which a relevant change occurs, i.e. the distance between two subsequent instructions in a test case is variable. In contrast, the test cases generated via TG-SRL always enumerate all discrete time steps, even for cases where no change happens. The distance between two subsequent instructions is always one time step.

For *Nat2Test*, it is possible to specify how many test cases shall be generated per requirement. However, with an increasing target, an exponential time to terminate the test generation has been observed. Thus, it was decided to adhere to a timeout of 3 hours per requirement. Increments of 10 were tested, and the results are as follows: for requirement 1 from Table 7.4, 40 test cases could be generated after approximately 130 seconds. With 50 test cases as a target, the tool ran into the specified timeout. For all other requirements, a timeout was observed with 40 as a target. Subsequently, for the other requirements 30 test cases have been generated for the evaluation.

the system mode	Idle: 1 Choice : 0 Strong: 2 Weak :3
the coffee machine output	Strong: 0 Weak : 1

Table 5.5: Enumeration in integer mapping for the *Vending Machine* system.

Time	the coffee re- quest button	the coin sen- sor	the system mode	the coffee ma- chine output
0.0	false	false	1	0
0.5	true	true	0	0
30.0	false	false	0	0
30.5	false	false	0	0
31.0	true	false	2	0
61.0	true	false	1	0

Table 5.6: Exemplary test case from *Nat2Test* for the *Vending Machine* system.

**Java Implementation** An implementation of the *Vending Machine* is available via [86]. However, due to the different approach to handling timers, this implementation does not fit both frameworks. In order to avoid having two different system implementations, which would negatively impact the comparability of the mutation testing results, an own implementation is used for both frameworks. With this implementation, the difference in handling timers is handled by the test engine instead. Listing 7.7 in the appendix shows the Java implementation. Notably, the boolean flag "isReset" has been introduced so that the test engine may know when to not increment the timer externally.

**Test Engine** A corresponding test engine, that executes and evaluates the test cases, has been implemented in Java. The concept is illustrated in Algorithm 8.

The increment of the timer in line 7 happens differently for both frameworks: with TG-SRL, the timer is incremented by 1 each time step as time is modeled discretely. In *Nat2Test*, the timer needs to be incremented by the time that has passed since the last reset. The concrete implementation is available in the appendix in Listings 7.1 and 7.2.

To validate that all generated test cases within a test suite adhere to the specification of the *Vending Machine*, the test suite has been executed with the original program with the expectation that all contained test cases pass. All generated test suites evaluated in this section fulfilled this criterion when executed on the original *Vending Machine*.



Time	the coffee machine output	the coffee request sensor	the sensor	coin	the request timer	the system mode
0	0	0	0		1	1
1	0	0	1		0	0
2	0	0	0		30	0
...	...	...	...		...	...
47	1	1	0		0	2
48	1	1	0		1	2
49	1	0	0		2	2
50	1	1	0		3	2
51	1	0	0		4	2
52	1	1	0		5	2
53	1	0	0		6	2

Table 5.7: Excerpt of an exemplary test case from TG-SRL for the *Vending Machine* system.

	Nat2Test	TG-SRL or tactic: (ext, ext) comp. tactic: (normal, normal) $d_w$ : 150
Requirements	5	5
Mutations	313	313
Test cases	160	18 (11 fail)
Time to generate tests	512s	69s
Mutants killed	236/313	270/313
Mutation score	75.4%	86.3%

Table 5.8: Mutation testing results for the *Vending Machine* system.

### 5.2.1 Mutant-Based Strength Analysis

The Java implementation has been mutated with *MuJava*, which was configured to apply all method-level operators. This resulted in 313 mutations. All mutations together with a unique ID are available in the Git repository [9].

Results from the initial evaluation are presented in Table 5.8. The time to execute and evaluate the tests via *MuJava* has been omitted as the time was always less than 3 minutes.

The set of alive mutants from this thesis’s framework is a subset of the alive mutants from *Nat2Test*. A manual analysis of all alive mutants revealed that 29 out of the 313 mutants were semantically equivalent to the original program. All the examples can be grouped into the following five categories.

**Algorithm 8** Test engine algorithm for the *Vending Machine* system.

---

```

1: read in all test cases from the test suite
2: for each test case do
3:   determine whether the test case is expected to fail via the name of the test case
4:   for every time step  $t$  within the test case do
5:     execute the SuT with the inputs specified at time  $t$ 
6:     if the timer has not been reset at time  $t$  then
7:       increment the timer
8:     end if
9:     update the variables "old coin sensor" and "old coffee request button"
10:    compare the actual outputs to the specified behavior: return "Fail" if deviations
        occur and the test case should pass
11:  end for
12:  if the test case passed for all time steps  $t$  but was expected to fail then
13:    return "Fail"
14:  end if
15:  return "Pass"
16: end for

```

---

**Category 1** List of mutants (2 in total):  $AOIS_{43}, AOIS_{44}$ 


---

```

1   if (a == y) then a = z
2   // modified to
3   if (a++ == y) then a = z
4   // or
5   if (a-- == y) then a = z

```

---

**Category 2** List of mutants (12 in total):  $CDL_{10}, CDL_{11}, CDL_{16},$   
 $CDL_{17}, CDL_7, CDL_8, ODL_{21}, ODL_{23}, ODL_{32}, ODL_{34}, ODL_{55}, ODL_{57}$ 


---

```

1   if (a == true)
2   // modified to
3   if (a)

```

---

**Category 3** List of mutants (8 in total):  $ODL_{16}, ODL_6, ROR_{24}, ROR_{26},$   
 $ROR_{27}, ROR_3, ROR_5, ROR_6$ 


---

```

1   // Modification of upper bound for request timer in a condition.
2   // Given the following condition:
3   if (timer >= lowerbound && timer <= upperbound && ...)
4   // the upper bound condition timer <= upperbound is either
        omitted, or modified to one of the following options:
5   timer == upperbound, timer < upperbound, timer != upperbound,
        true

```

---

---

```

1  if ((the_request_timer <= 30.0) && (the_request_timer >= 10.0) &&
    (the_system_mode == 3))
2  // modified to
3  if ((the_request_timer < 30.0) && (the_request_timer >= 10.0) &&
    (the_system_mode == 3))

```

---

Listing 5.1: Mutant  $ROR_3$ **Category 4** List of mutants (2 in total):  $ROR_{11}$ ,  $ROR_{32}$ 


---

```

1  // Modification of lower bound for request timer in a condition.
2  // Given the following condition:
3  if (timer >= lowerbound && timer <= upperbound && ...),
4  // the lower bound condition timer >= lowerbound is modified to:
5  timer == lowerbound

```

---

**Category 5** List of mutants (5 in total):  $ROR_{74}$ ,  $ROR_{78}$ ,  $ROR_{79}$ ,  $ODL_{72}$ ,  $ROR_{37}$ 


---

```

1  if (a) then (b) else if (not a) then (c)
2  // modified to
3  if (a) then (b) else if (true) then (c)

```

---

Both categories 1 and 2 are equivalent due to properties of the Java language. 1 is semantically equivalent due to the sequence of operations in Java. The postfix operator is applied after the if statement has been applied, i.e. the variable "a" is incremented or decremented by 1. However, directly thereafter a fixed value is assigned to variable "a", thus invalidating the modification from the postfix operator. The modification in category 2 is equivalent because the statements "if (x)" and "if (x==true)" are semantically equivalent in Java.

For categories 3, 4, and 5, the semantic equivalence arises due to properties of the *Vending Machine* system. To illustrate category 3, consider the concrete mutant  $ROR_3$  in Listing 5.1.

To meet the condition, the request timer needs to be within the interval 10 to 30 and the system mode needs to be equal to 3. Both signals mentioned in this condition are output signals and cannot be modified by e.g. a user. Thus, it is necessary to investigate in which scenarios the system mode can become 3. As the initial value of the system mode is 1, it is necessary to look at all assignments to 3. This is only the case for a single line, namely line 35 in Listing 7.7. If the corresponding condition becomes true, the request timer is also simultaneously reset to 0 (line 33). Subsequently, it then takes several time steps for the timer to fulfill the condition " $\geq 10$  and  $\leq 30$ ". But notably, this condition is always triggered as soon as the timer reaches the value 10, as no external source can modify the system mode to a value other than 3 while the timer increments. This is illustrated in Table 5.9. Concluding, the mutation is semantically equivalent to

---

```

1  if (timer <= 30) then <actions>
2  else if (timer > 30) then <actions>
3  // can be simplified to:
4  if (a) then (b) else if (not a) then (c)
5  // which is equivalent to:
6  if (a) then (b) else if (true) then (c)

```

---

Listing 5.2: Illustration of semantic equivalence for the mutations in category 5 from Section 5.2.1.

the original program. The same argumentation can be applied for all mutants in this category.

Time	Signal values	Observation
$t$	timer = 0, mode = 3	
$t + 1$	timer = 1, mode = 3	
$t + 10$	timer = 10, mode = 3	triggers condition, mode will be set to 1
$t + 10$	timer = 10, mode $\neq$ 3	cannot happen because mode cannot be modified externally
$t + 11$	timer = 11, mode = 3	cannot happen because the condition will always be triggered in the previous step and will modify mode to 1

Table 5.9: Illustration why the condition "timer  $\geq$  10 && timer  $\leq$  30 && mode == 3" will always be triggered with a timer value of 10.

Category 4 is very similar with the difference being that the lower bound is modified from a " $\geq$ " to " $=$ ". Mutations in category 5 are semantically equivalent due to the if else construct used in the implementation in lines 30-43 in Listing 7.7. Listing 5.2 illustrates this equivalence.

The resulting mutation scores when taking semantic equivalence into account are presented in Table 5.10.

**Analysis of alive Mutants** In the following, all alive mutants are analyzed to understand which cases are not covered by the generated test suite of TG-SRL. In  $SDL_7$ , an assignment of an output signal is completely removed. This is illustrated in Listing 5.3.

As explained in Section 4.5, for every condition, a test case is generated that enforces this condition at a certain time point  $t \in \mathbb{N}_0$ . The fact that such a test case did not catch the removal of "b" implies that "b" had the same value in the previous iteration. This can occur if a test case does not modify a signal at all, such that the signal retains its initial value throughout the whole test case. Indeed, in  $SDL_7$  a statement is removed

Nat2Test		<b>TG-SRL</b> or tactic: (ext, ext) comp. tactic: (normal, normal) $d_w$ : 150
Requirements	5	5
Mutations	284	284
Test cases	160	18 (11 fail)
Time to generate tests	512s	69s
Mutants killed	236/284	270/284
Mutation score	83.1%	95.1%

Table 5.10: Mutation testing results for the *Vending Machine* system after accounting for semantically equivalent mutants.

```

1 if (cond) then (a && b)
2 // modified to
3 if (cond) then (a)

```

Listing 5.3: Exemplary mutation that removes an action completely.

that would assign the "coffee machine output" the value 0, which corresponds to its initial value.

To cover such cases, the algorithm to generate a test suite in Listing 7 has been extended with the option to enforce a change of the initial value at a certain time point within the test case. Assuming an initial value of 5 for a signal "x", and that the antecedent is enforced to be true at time point  $t \in \mathbb{N}_0$ , the following constraints are generated:

$$\begin{aligned}
& x(0) = 5 \\
& \exists t_x \text{ with } x(t_x) \neq 5 \text{ and } 0 < t_x < t
\end{aligned}$$

With this change, four additional mutants are killed. This impacts the mutation score as shown in Table 5.11. Newly killed mutants are as follows:  $SDL_7$ ,  $ODL_{45}$ ,  $ODL_{64}$ ,  $ODL_{68}$ . In  $ODL_{45}$ ,  $ODL_{64}$ , and  $ODL_{68}$  a condition referring to the "coin sensor" has been removed completely. In all cases, it was checked whether the coin sensor is or was "false". As "false" is the initial value for the "coin sensor", these mutants are also only killed by enforcing an initial value change.

The remaining 10 mutants that are left alive can be put into three categories:

**Category 1** Conditions modified so that out of bound values are referenced. List of mutants (3 in total):  $ROR_{16}$ ,  $ROR_{66}$ ,  $ROR_{84}$

**Category 2** Condition "isReset = false" removed. List of mutants (2 in total):  $SDL_4$ ,  $SDL_9$

	Nat2Test	TG-SRL or tactic: (ext, ext) comp. tactic: (normal, normal) initial value change: true $d_w$ : 150
Requirements	5	5
Mutations	284	284
Test cases	160	18 (11 fail)
Time to generate tests	512s	69s
Mutants killed	236/284	274/284
Mutation score	83.1%	96.5%

Table 5.11: Mutation testing results for the *Vending Machine* system after enforcing changes of the initial value for a signal.

```

1  if (coin sensor = true && old coin sensor = false && system mode =1) then
2      ...
3  // modified to
   if (coin sensor = true && system mode =1) then ...

```

Listing 5.4: Exemplary mutation that omits a condition completely.

**Category 3** Condition has been modified. Behavior unspecified for the modification. List of mutants (5 in total):  $ODL_{26}$ ,  $ODL_{60}$ ,  $ODL_{76}$ ,  $ROR_{82}$ ,  $ROR_{86}$

In category 1, modifications are done that cannot be detected due to constraints that enforce upper and lower boundaries for signals. For instance, the "system mode = 3" is modified to "system mode  $\geq$  3" in line 15 of Listing 7.7. No test case generated via the SMT solver will contain the value 4 for "system mode" as another upper bound constraint enforces all "system mode" values to be lower or equal to 3.

In category 2, the "isReset = false" statement is removed. In the implementation of the test driver shown in Listing 7.1, this flag is used to decide whether to increment the request timer. However, only in very few cases "isReset" is "true". Thus, a test case would need to enforce a state where "isReset" is "true", afterwards enforce the state where the "isReset = false" statement has been removed, and lastly enforce a state where the missing increment of the timer is causing erroneous behavior. The generated test cases from both frameworks do not reach this complexity. For the SMT instances presented in this thesis, test cases are generated with the constraint of enforcing one antecedent to become true at a specific point in time. In contrast, in the scenario described above, three antecedents need to be enforced in the proper sequence during one test cases.

In category 3, conditions are modified or omitted completely. Listing 5.4 illustrates this.

Given this concrete example, it is evident that no test case sets the "coin sensor" to "true", the "old coin sensor" to "true", and the "system mode" to 1. If such a test case

would exist, the mutated program would trigger the corresponding actions, which is not expected to happen according to the specification.

In order to address these cases, the test suite generation algorithm from Listing 7 has been extended to generate variants for antecedents. As specified in Listing 4.1, antecedents are conjunctions of boolean expressions. Given a conjunction  $\wedge(x_1(t), \dots, x_n(t))$ ,  $n$  additional test cases are added to the test suite with the following modified antecedent: one term  $x_i$  is negated while the other terms remain as is. For the conjunction  $\wedge(x_1(t), x_2(t), x_3(t))$ , this strategy results in three additional expressions:

- $\wedge(\neg x_1(t), x_2(t), x_3(t))$
- $\wedge(x_1(t), \neg x_2(t), x_3(t))$
- $\wedge(x_1(t), x_2(t), \neg x_3(t))$

Similar to before, for each antecedent variation, one SMT instance is created where the specific antecedent is enforced at a time point  $t \in \mathbb{N}_0$ . With this extension, all mutants from category 3 are killed. Table 5.12 summarizes the final mutation testing results for the *Vending Machine* system.

Nat2Test		<b>TG-SRL</b> or tactic: (ext, ext) comp. tactic: (normal, normal) initial value change: false and combinations: false $d_w$ : 150	<b>TG-SRL</b> or tactic: (ext, ext) comp. tactic: (normal, normal) initial value change: true and combinations: false $d_w$ : 150	<b>TG-SRL</b> or tactic: (ext, ext) comp. tactic: (normal, normal) initial value change: true and combinations: true $d_w$ : 150
Requirements	5	5	5	5
Mutations	284	284	284	284
Test cases	160	18 (11 fail)	18 (11 fail)	37 (11 fail)
Time to generate tests	512s	69s	69s	161s
Mutants killed	236/284	270/284	274/284	279/284
Mutation score	83.1%	95.1%	96.5%	98.2%

Table 5.12: Mutation testing results for the *Vending Machine* system.

### 5.3 Daimler Turn Indicator System

The third evaluated system is a *Turn Indicator* example from Daimler. Requirements have been taken from the *Nat2Test* website [86].

The system is specified by 17 requirements in total. The requirements in SysReq-CNL can be found either on the corresponding website [86] or in the Git repository for this thesis [9]. The preprocessed requirements used as an input for TG-SRL are available in Table 7.5 in the appendix. In Table 5.13 the input and output signals are specified.

Signal	Type	Datatype	[Initial Lower Bound, Upper Bound]	Value, Bound, Upper Bound]
the voltage	input	number	[0,0,100]	
the turn indicator lever	input	number	[0,0,2]	
emergency mode	input	bool	[0,0,1]	
the indication lights	output	number	[2,0,3]	
the mode	output	number	[2,0,3]	
the timer	output	timer	[0,0,100]	

Table 5.13: Input and output signals from the *Turn Indicator* system.

One notable change to the requirements for TG-SRL is the modification of the timer thresholds. In multiple requirements, a condition is specified that holds true if the "flashing timer" is greater or equal to 340 or 220. In order to reduce the solution space for the SMT solver, these values have been modified to 34 and 22, respectively. Without this modification, every constraint that is valid for a time step  $t$  would be enumerated 340 times due to universal quantifier elimination (see Section 4.5.1). This in turn would negatively impact the time to generate the test suite.

Similar to the *Vending Machine*, the *Turn Indicator* system also refers to a timer. Another particularity is that, given a time slot  $t$ , multiple conditions might hold true. This results in the following issue:

**Requirements:**

$s_1 := \text{if } a(t) \text{ then } b(t) = 5$

$s_2 := \text{if } b(t_1) = 5 \text{ then } c(t_1) = 5$

Given " $a(t) = \text{true}$ " at time point  $t$ , it is subject to interpretation whether the condition " $b(t_1) = 5$ " evaluates to true in the same time step  $t$ . Furthermore, during the system's implementation, the sequence of operations naturally establishes a priority. If the statement  $s_2$  precedes  $s_1$ ,  $s_2$  will not take effect.

The framework presented in this thesis assumes that  $s_2$  can only occur at the next time step because  $s_1$  already modifies the signal " $b(t)$ " at time  $t$ . Conversely, *Nat2Test* assumes that both conditions can become true at the same time. In both instances, this behavior needs to be accounted for during the implementation. This is achieved via the introduction of temporary variables that decouple a condition from an assignment, as illustrated in Listing 5.5.

The difference in interpretation between both approaches necessitates slightly different implementations of the *Turn Indicator* system. The implementations are available in the appendix in Listings 7.3 and 7.4. The evaluation is not impacted significantly as the structure of both programs is equivalent and thus, by applying mutation operators, similar mutants are created.



---

```

1  if (a) then  $b_{temp}$ 
2  if (b) then c
3  b =  $b_{temp}$ 

```

---

Listing 5.5: Illustration of decoupling two conditions that can become true in the same time step.

Nat2Test		<b>TG-SRL</b> or tactic: (ext, ext) comp. tactic: (normal, normal) initial value change: false and combinations: false $d_w$ : 150	<b>TG-SRL</b> or tactic: (ext, ext) comp. tactic: (normal, normal) initial value change: true and combinations: true $d_w$ : 150
Requirements	17	17	17
Mutations	1101	1101	1101
Test cases	510	190 (100 fail)	237 (98 fail)
Time to generate tests	255m	16m	21m
Mutants killed	661/1101	868/1101	876/1101
Mutation score	60.0%	78.8%	79.6%

---

Table 5.14: Mutation testing results for the *Turn Indicator* system.

Similar to the *Vending Machine* system, the output values are expected to be held if no condition triggers a modification of the values at a certain time point  $t$ . Thus, the optional constraints described in Section 4.5.1 were also employed for this system.

All generated test cases for the *Turn Indicator* system for both approaches can be found in the Git repository for this thesis [9]. For the worst-case upper bound  $d_w$  a value of 150 is used. In the modified requirements, conditions only refer to timer values of 34, thus 150 provides sufficient overhead.

For *Nat2Test*, the same approach to test generation has been chosen as with the *Vending Machine*. In the case of the *Turn Indicator* system, timeouts occurred with 40 test cases. Thus, for each requirement, 30 test cases have been generated, which results in 510 test cases in total.

The test engines implemented for both frameworks follow the same concept as described in Section 5.2 and can be found in the appendix in Listings 7.5 and 7.6. As with previous systems, all test suites have been run on the original program in order to validate them.

### 5.3.1 Mutant-Based Strength Analysis

*MuJava* was used with all method-level operators enabled and resulted in 1101 mutations. The mutations are available in the Git repository [9]. The results are given in Table 5.14. For the *Turn Indicator* system, no manual analysis of semantic equivalence has been done on the alive mutants. Thus, Table 5.14 presents worst-case mutation scores.

## 5.4 Evaluation of Objectives

In the introduction of this thesis in Section 1.1, several objectives were formulated. In this section, the objectives are evaluated briefly.

**Objective:** Provide a proof of concept for a test generation framework that operates directly on natural language requirements. The input domain should be as unconstrained as possible, i.e., restrictions on the input language should be kept to a minimum.

**Evaluation:** This objective is only partially fulfilled. The presented framework succeeds in generating a test suite based on natural language requirements. However, there are significant assumptions made about the input domain, e.g., with regards to the level of detail in the requirement set. The use of a rule-based approach to extract information and formalize the extracted information is limited conceptually, as only cases can be covered which are covered by the implemented ruleset. These limitations make preprocessing of the requirements a necessity, thus constraining the input domain. Suggestions have been given on how to generalize the approach. For instance, the implemented ruleset currently limits the allowed vocabulary in requirements. This can be tackled by looking up synonyms prior to extracting information in Stage 3, or extending the supported predicates to those available in the ProBank model.

Other limitations that restrict the input domain, such as the algorithms to infer the logical connection between entities, are harder to solve. Such tasks are best tackled via NLP methods that rely on machine learning, as a rule-based approach has difficulties covering the complex cases that may occur in natural language. To the best of the author’s knowledge, no NLP method exists currently that succeeds in properly extracting logical relationships between entities in natural language.

**Objective:** Achieve the highest automation degree possible: avoid the manual creation of dictionaries or domain knowledge, and avoid manual steps within the automation process. The generated test cases should be executable without manual refinement.

**Evaluation:** Assuming a priori knowledge of the signal specification is given, the presented framework does not entail any manual steps apart from the preprocessing of the natural language requirements, and the estimation of a worst-case upper bound for the test duration. In particular, compared to existing approaches described in Chapter 3, it is not necessary to solve ambiguity in a manual way, or to manually translate abstract test cases into executable ones. Knowledge about in- and output signals can theoretically be extracted automatically from a natural language specification, or a data dictionary. The preprocessing of the requirements will become less relevant with increasing quality and capabilities of NLP and SRL frameworks, assuming the respective rules are also extended. Thus, the presented concept achieves a significant automation degree.

**Objective:** Optimize the strength of the test suite while accounting for test suite size: maximize the test suite’s capability to detect system behavior that deviates from the specification, and keep its size within reasonable boundaries.

**Evaluation:** This objective is fulfilled. Several tactics have been implemented to improve the test suite’s strength. During the design of these tactics, test suite size was considered. Additionally, several measures such as quantifier elimination and the minimization of the test duration have been implemented to improve the performance of the test suite. In a comparison with the framework *Nat2Test*, TG-SRL performs favorably in a mutation-based testing analysis while featuring significantly smaller test suites.

## 5.5 Limitations & Future Work

In TG-SRL, the employed NLP methods such as SRL do not extract sufficient information for test generation. Thus, these methods are supplemented with heuristics and a set of rules, which inherently limit the input domain. In the following, a subset of the resulting limitations and respective opportunities for future work are enumerated.

1. The currently employed SRL models have not been trained for the domain at hand. Training on domain-specific requirements from embedded systems has the potential to significantly improve the quality of the SRL output, and thus reduce the manual preprocessing effort.
2. The current implementation looks at single requirements in isolation and does not consider references or context from other requirements. NLP methods from the area of discourse semantics or language modeling may be employed to extract information from complete requirement sets instead of individual requirements.
3. The currently implemented rule-set only covers a specific set of verbs and language constructs that are used to specify signal behavior. For instance, the requirement "A signal ramps from *<value1>* to *<value2>* within *<time>*." cannot be processed. Future work could look into implementing rules for the complete list of verbs from ProbBank.
4. The currently implemented rule-set does not support variables or enumerations as signal values.
5. The implemented heuristic to extract the logical connection between entities is restrictive and cannot cover all cases that may be found in natural language requirements. Instead of utilizing a heuristic, methods from the area of language modeling may be employed.
6. The formalization of the requirements as an SMT instance limits the mathematical complexity that requirements may contain. For example, with non-linear mathematical constraints in requirements, an SMT solver is less likely to find a satisfying solution and thus generate a test case.

7. Ambiguity is in many cases resolved through the rule-set. There is a significant risk that the chosen interpretation deviates from that of the requirements author. Instead of using a rule-set, Large Language Models (LLMs) might be better suited to determine the most probable interpretation.

Apart from leveraging other NLP methods, there is also potential to look at extensions for SRL. SRL as presented in this framework identifies semantic roles for predicates. However, there is also research that extends the semantic analysis to nouns and prepositions [80, 81]. This enables a more precise description of the roles relative to the sense of the predicate as illustrated in Figure 5.6

Furthermore, when comparing TG-SRL to the approach from Schnelte et al. (refer to Section 3.3.1), another possibility for future work is revealed. Assuming certain conditions are met and a system reaction is expected, the approach from Schnelte et al. can deal with a reaction occurring during a given time interval. In contrast, the currently implemented proof of concept only supports reactions that happen at a deterministic point in time. Even if left unspecified, the implemented ruleset assumes that the action happens in the same time step  $t$  in which the conditions were met. A possible extension could look as follows: by implementing a rule to support the notion of "action  $a$  happens within  $x$  seconds", one could support an unknown response time of a system. A formalization into FOL could look as follows:

$$\exists t_1 \text{ with } \textit{action } a = \textit{true} \text{ and } t < t_1 < t + x$$

### 5.5.1 Large Language Models

Recent advancements in Large Language Models (LLM) offer a promising outlook to improve test generation based on unconstrained natural language requirements. In the context of this thesis, LLMs may be utilized in one of the following ways.

1. Utilization of LLMs for both information extraction and test generation
2. Utilization of LLMs for information extraction, i.e. replacing or improving the information extraction via SRL and the rule-set
3. Utilization of LLMs to improve individual components of TG-SRL, e.g., by resolving ambiguity or by improving the identification of conditional and non-conditional parts of a sentence

An initial experiment with GPT-4o from OpenAI [88][89], carried out in January of 2025, is described in the following. The results are promising and suggest that LLMs may be suited to achieve both information extraction and test generation at once. As an input to the model, the following prompt was specified:

*I have a natural language requirement and would like to generate executable test cases out of it. A test case consists of time value pairs, e.g. at time step 0*

*signal A has the value 1. At time step 1, signal A may have a value 2. Please generate test cases for the following requirement: "If the flashing timer is greater than 20, then increment the speed by 10 after 3 seconds and reset the timer to 0."*

GPT-4o's response is shown in Figure 5.7. It successfully generates three test cases, one covering a scenario where the requirement is met, one where the requirement is not met, and one that tests an edge case. It is noteworthy that the model's interpretation of "increment the speed by 10 after 3 seconds" differs from the interpretation of TG-SRL. Assuming  $t$  as the current time step, GPT-4o changes the signal value of the signal *speed* at time  $t+4$  while TG-SRL enacts this signal change at time  $t+3$ . Depending on the requirement author's intention, prompt engineering and in particular system prompts could be utilized to modify this interpretation for an LLM.

Figure 5.8 showcases how LLMs may be utilized to improve individual components of TG-SRL. In the given example, GPT-4o resolves the ambiguity with regards to operator precedence. Compared to the implemented rule-set in TG-SRL, an approach utilizing LLMs may pick up nuances in the natural language sentences and select the most probable interpretation accordingly. Nevertheless, the example also illustrates that the output of an LLM is still error-prone: although the requirement specifies "signal  $d$  is [...] equal to 5", the logical representation from GPT-4o specifies "equal or greater to 5" and the given explanation contains the condition "signal  $d$  is [...] at least equal to 5".

Apart from the above mentioned examples, the saved GPT-4o dialogue provided in [89] also showcases that an LLM is capable of handling mathematical operations within requirements and performing coreference resolution between multiple requirements.

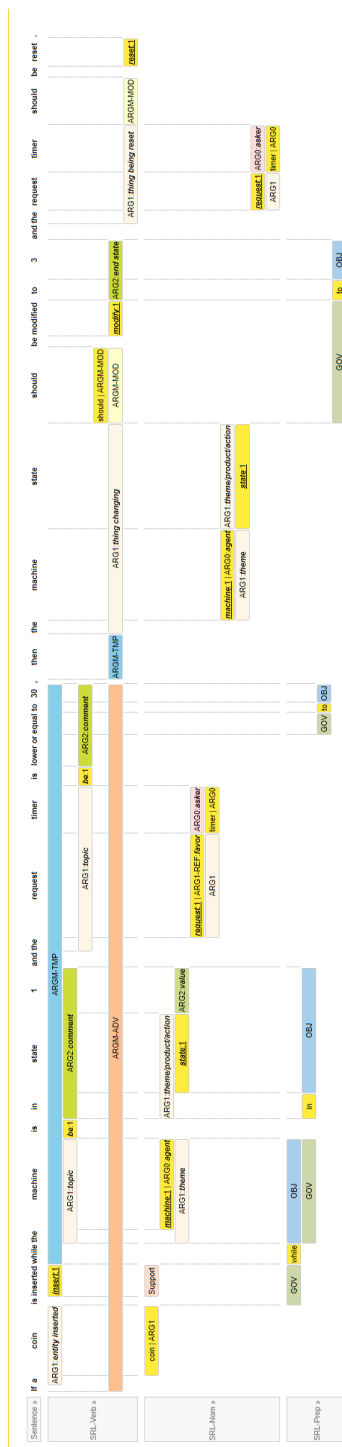


Figure 5.6: SRL-Verb, SRL-Nom and SRL-Prep output from [76].

**Requirement:**

"If the flashing timer is greater than 20, then increment the speed by 10 after 3 seconds and reset the timer to 0."

**Test Case Generation:****Signals:**

- **Flashing Timer:** Tracks the time since the timer was started.
- **Speed:** Tracks the speed of the system.

**Test Case 1: Positive Scenario (Requirement met)**

**Initial Condition:** Flashing timer > 20 at some time step.

**Expected Behavior:** After 3 seconds, the speed is incremented by 10, and the flashing timer resets to 0.

Time Step	Flashing Timer	Speed	Comments
0	21	50	Flashing timer > 20; speed is 50
1	22	50	Timer continues incrementing
2	23	50	Timer continues incrementing
3	24	50	Timer continues incrementing
4	0	60	Speed incremented, timer reset

**Test Case 2: Negative Scenario (Requirement not met)**

**Initial Condition:** Flashing timer ≤ 20 at all times.

**Expected Behavior:** No change to speed, and the timer does not reset.

Time Step	Flashing Timer	Speed	Comments
0	10	50	Flashing timer ≤ 20
1	11	50	Timer continues incrementing
2	12	50	Timer continues incrementing
3	13	50	Timer continues incrementing
4	14	50	No change

**Test Case 3: Edge Case (Boundary Value)**

**Initial Condition:** Flashing timer exactly 20.

**Expected Behavior:** No change to speed or timer since the requirement specifies "greater than 20."

Time Step	Flashing Timer	Speed	Comments
0	20	50	Flashing timer = 20
1	21	50	Timer starts incrementing
2	22	50	Timer continues incrementing
3	23	50	Timer continues incrementing
4	24	50	No change

Figure 5.7: Generated test cases from GPT-4o [89] on the given input requirement.

Please show me a logical representation of the following requirement:  
"If signal d is true, then signal a and signal b is true or signal c is true,  
and signal d is greater than 10 or equal to 5."



The logical representation of the requirement:

"If signal d is true, then signal a and signal b is true or signal c is true, and signal d is greater than 10 or equal to 5."

Can be expressed in first-order logic as follows:

#### Logical Representation

$$d \implies ((a \wedge b) \vee (c \wedge (d \geq 5 \vee d > 10)))$$

#### Explanation:

- $d$  is the condition that triggers the implication.
- $(a \wedge b)$  represents that both signal `a` and signal `b` must be true.
- $(c \wedge (d \geq 5 \vee d > 10))$  represents that signal `c` must be true, and signal `d` must be either greater than 10 or at least equal to 5.

Figure 5.8: Example that illustrates how GPT-4o [89] can be utilized to resolve ambiguity in a given requirement.



## 6 Conclusion

In this thesis, the framework TG-SRL is introduced. TG-SRL tackles the challenge of test generation based on natural language requirements by combining Natural Language Processing (NLP) methods with a rule-based approach. In particular, Semantic Role Labeling (SRL) is employed to identify predicate argument structures of a requirement, and extract semantic information. The rule-based stage then aggregates the extracted information into logical expressions with temporal properties. These in turn are formalized into a First-Order Logic (FOL) representation, where time is encoded symbolically. With the help of a Satisfiability Modulo Theory (SMT) solver, executable test cases are generated. By modifying the SMT instance via several proposed tactics, a strong test suite can be derived.

An evaluation of the approach has been conducted on three systems. In a mutation-based strength analysis, TG-SRL has been compared to the framework *Nat2Test* [8, 70, 17] and performed favorably. TG-SRL is highly automated, produces executable test cases and can operate on almost unrestricted natural language requirements. Inter alia due to the maturity of the employed NLP methods and the prototypical nature of the rule-based stages, in practice manual preprocessing of the requirements is necessary.

Concluding, the presented approach provides valuable insights into employing NLP methods, and in particular Semantic Role Labeling, in the field of test case generation, showcasing their potential for future work in this area.

## 6 Conclusion

## 7 Appendix

ID	Requirement
1	When <i>driver_up</i> , <i>driver_down</i> , <i>passenger_up</i> and <i>passenger_down</i> is false, the system shall set <i>move_up</i> to false, assign false to <i>move_down</i> and assign 0 to <i>error_counter</i> .
2	When <i>driver_up</i> and <i>driver_down</i> is true and <i>passenger_up</i> and <i>passenger_down</i> is true, then the system shall set <i>error_counter</i> to 2.
3	When <i>driver_up</i> and <i>driver_down</i> is true and <i>passenger_up</i> or <i>passenger_down</i> is false, then the system shall set <i>error_counter</i> to 1.
4	When <i>passenger_up</i> and <i>passenger_down</i> is true and <i>driver_up</i> or <i>driver_down</i> is false, then the system shall set <i>error_counter</i> to 1.
5	When <i>driver_up</i> or <i>driver_down</i> is false and <i>passenger_up</i> or <i>passenger_down</i> is false, then the system shall set <i>error_counter</i> to 0.
6	When <i>error_counter</i> is 0, then <i>driver_up</i> or <i>driver_down</i> is false and <i>passenger_up</i> or <i>passenger_down</i> is false.
7	When <i>error_counter</i> is greater than 0, then <i>driver_up</i> and <i>driver_down</i> is true or <i>passenger_up</i> and <i>passenger_down</i> is true.
8	When <i>driver_up</i> is true and <i>error_counter</i> is 0 and <i>obstacle_detection</i> is false, then the system shall assign true to <i>move_up</i> .
9	When <i>passenger_up</i> is true and <i>driver_down</i> is false and <i>error_counter</i> is 0 and <i>obstacle_detection</i> is false, then the system shall assign true to <i>move_up</i> .
10	When <i>move_up</i> is true, then <i>driver_up</i> or <i>passenger_up</i> is true and <i>error_counter</i> is 0 and <i>obstacle_detection</i> is false.
11	When <i>driver_down</i> is true and <i>error_counter</i> is 0, then the system shall assign true to <i>move_down</i> .
12	When <i>passenger_down</i> is true and <i>driver_up</i> is false and <i>error_counter</i> is 0, then the system shall assign true to <i>move_down</i> .
13	When <i>move_down</i> is true, then <i>driver_down</i> or <i>passenger_down</i> is true and <i>driver_up</i> is false and <i>error_counter</i> is 0.
14	When <i>move_up</i> is true, then <i>move_down</i> is false.
15	When <i>move_down</i> is true, then <i>move_up</i> is false.
16	When <i>passenger_up</i> is true and <i>driver_down</i> is true and <i>error_counter</i> is 0, then <i>move_down</i> is true.
17	When <i>passenger_down</i> is true and <i>driver_up</i> is true and <i>error_counter</i> is 0 and <i>obstacle_detection</i> is false, then <i>move_up</i> is true.
18	When <i>error_counter</i> is greater than 0, then <i>move_down</i> is false and <i>move_up</i> is false.

19	When <i>driver_information</i> is 2, then <i>error_counter</i> is greater or equal to 1.
20	When <i>driver_information</i> is 1, then the <i>vehicle_speed</i> is greater than 30, and <i>current_position</i> is less than 100, and the <i>error_counter</i> is 0.
21	When <i>driver_information</i> is 0, then <i>error_counter</i> is 0 and the <i>vehicle_speed</i> is less or equal to 30 or <i>current_position</i> is 100.
22	When <i>obstacle_detection</i> is true, then the system shall set <i>move_up</i> to false.
23	When <i>current_position</i> is less than 100 and <i>move_up</i> is true, then the system shall add 1 to <i>current_position</i> .
24	When <i>move_up</i> is true and <i>current_position</i> is 100, then <i>current_position</i> is 100.
25	When <i>move_down</i> is true and <i>current_position</i> is higher than 0, then the system shall subtract 1 from <i>current_position</i> .
26	When <i>move_down</i> is true and <i>current_position</i> is 0, then <i>current_position</i> is 0.
27	When <i>current_position</i> changes its value, then <i>move_up</i> is true, or <i>move_down</i> is true.
28	When <i>current_position</i> changes its value, then the system shall subtract 1 from <i>current_position</i> or it shall add 1 to <i>current_position</i> .

Table 7.1: List of requirements for the *Window Control* system.

<b>Subsystem:</b> WindowControl/Variant Subsystem/window control1/signal creator/driver signals/validate driver signals	
ID	Description
1	"++" to "+-": no error is generated when both <i>driver_up</i> and <i>driver_down</i> signals are simultaneously 1.
2	"2 -> 1": an error is triggered if <i>driver_up</i> or <i>driver_down</i> are true on their own or together, no error is triggered if both are false.
3	">=" to "<": an error is triggered if <i>driver_up</i> or <i>driver_down</i> are true alone or both are false, no error is triggered if both are true.
<b>Subsystem:</b> WindowControl/Variant Subsystem/window control4/signal creator/driver signals/check driver control	
ID	Description
4	"1 -> 3": <i>move_up</i> is not activated when <i>driver_up</i> equals 1, <i>move_down</i> is not activated when <i>driver_down</i> equals 1.
<b>Subsystem:</b> WindowControl/Variant Subsystem/window control5/check errors/pass driver signals	
ID	Description
5	"0->1": only when two errors occur <i>move_up</i> or <i>move_down</i> are prevented.
6	"0-> -1": even when no error occurs, <i>move_up</i> or <i>move_down</i> are prevented.
7	Switch for <i>up signal</i> removed: regardless of whether errors occur, if <i>driver_up</i> , then <i>move_up</i> .
8	Switch for <i>down signal</i> removed: regardless of whether errors occur, if <i>driver_down</i> , then <i>move_down</i> .

9	<i>in_up</i> and <i>in_down</i> swapped: if <i>driver_up</i> then <i>move_down</i> and if <i>driver_down</i> then <i>move_up</i> .
<b>Subsystem:</b> WindowControl/Variant Subsystem/window control9/active signal filter	
ID	Description
10	"0 -> 1": even if an obstacle has been detected, <i>move_up</i> is true. If <i>passenger_up</i> and <i>driver_up</i> are both false, <i>move_up</i> is still true. If <i>passenger_down</i> and <i>driver_down</i> are both false, <i>move_up</i> is true.
11	Obstacle check removed: even if <i>obstacle_detected</i> is true (and <i>driver_up</i> or <i>passenger_up</i> are true), <i>move_up</i> is true.
12	">0" check connected with <i>passenger_active</i> : even if <i>driver_down</i> equals false and <i>driver_up</i> equals true, <i>passenger_down</i> takes priority, i.e., <i>passenger_down</i> equals true and <i>up</i> equals false leads to <i>move_down</i> and <i>move_up</i> simultaneously.
13	">0" check connected with <i>passenger_active</i> instead of <i>driver_active</i> : as in 12, but reversed.
<b>Subsystem:</b> WindowControl/Variant Subsystem/window control13/error sum	
ID	Description
14	"++" to "+-": if <i>driver_up and down</i> equals true and <i>passenger_up and down</i> equals true, then the error counter is 0 instead of 2 and thus <i>driver_info</i> is not 2. If there is only a <i>passenger_error</i> , the error sum is -1 and thus <i>driver_info</i> is 0.
<b>Subsystem:</b> WindowControl/Variant Subsystem/window control14/window position	
ID	Description
15	Up and down direction swapped: if <i>move_up</i> is true, then <i>current_position</i> is decremented and vice versa.
16	Set the else case to <i>direction = 1</i> : if neither <i>move_up</i> nor <i>move_down</i> is true, then <i>current_position</i> is decremented.
17	"-1" to "-2" for down: if <i>move_down</i> is true, then <i>current_position</i> changes by 2 instead of 1.
18	Inserted delay after the multiswitch: if <i>move_down</i> or <i>move_up</i> is true, then <i>current_position</i> only changes in the next time step.
19	The upper bound of <i>current_pos</i> in the saturate block has been set to 150.
20	The lower bound in the saturate block has been set to 50.
<b>Subsystem:</b> WindowControl/Variant Subsystem/window control20/driver information system	
ID	Description
21	"error > 0" to "error > 1": if there is an error, <i>driver_information</i> is not 2
22	"error > 0" to "error >= 0": even if there is no error, <i>driver_info</i> is 2.
23	When there is an error, <i>driver_info</i> is 1 instead of 2.
24	"vehicle speed > upper vs" to ">=": when speed equals 30, then <i>driver_info</i> is 1 if energy can be saved.
25	"vehicle speed > upper vs" to "<": if speed is less than 30 (instead of >30), <i>driver_info</i> is 1
26	"position <100" to "<=": even when the window is already closed ( <i>position</i> = 100), <i>driver_info</i> is 1.

27	<i>Driver_info</i> is 3 when energy can be saved (instead of 1).
28	Driver info = 1 instead of 0 at the start: even if there is no error and no energy can be saved, <i>driver_info</i> is 1.
<b>Subsystem:</b> WindowControl/Variant Subsystem/window control5/check errors	
ID	Description
29	"++" block removed: if <i>passenger_up</i> , <i>down</i> equals true and an error has been caused, then for <i>driver_up</i> or <i>down</i> equals true, <i>move_up</i> or <i>down</i> is still true.
30	"++" to "+-": if <i>passenger_up</i> , <i>down</i> and <i>driver_up</i> , <i>down</i> are all true, then <i>move_up</i> and <i>down</i> are set to true simultaneously.

Table 7.2: Mutations of the *Window Control* system

ID	Requirement
1	When the <i>system mode</i> is 1, and the <i>coin sensor</i> is on, and the <i>coin sensor</i> was off, the coffee machine system shall set the <i>system mode</i> to 0 and reset the <i>request timer</i> .
2	When the <i>system mode</i> is 0, and the <i>coffee request sensor</i> is on, and the <i>coffee request sensor</i> was off, and the <i>coin sensor</i> is off, and the <i>coin sensor</i> was off, and the <i>request timer</i> is lower or equal to 30, the coffee machine system shall set the <i>system mode</i> to 3 and reset the <i>request timer</i> .
3	When the <i>system mode</i> is 0, and the <i>coffee request sensor</i> is on, and the <i>coffee request sensor</i> was off, and the <i>coin sensor</i> is off, and the <i>coin sensor</i> was off, and the <i>request timer</i> is greater than 30, the coffee machine system shall set the <i>system mode</i> to 2 and reset the <i>request timer</i> .
4	When the <i>system mode</i> is 3, and the <i>request timer</i> is greater or equal to 10, and the <i>request timer</i> is lower or equal to 30, the coffee machine system shall set the <i>system mode</i> to 1 and it shall set the <i>coffee machine output</i> to 1.
5	When the <i>system mode</i> is 2, and the <i>request timer</i> is greater or equal to 30, and the <i>request timer</i> is lower or equal to 50, the coffee machine system shall set the <i>system mode</i> to 1 and it shall set the <i>coffee machine output</i> to 0.

Table 7.3: Preprocessed list of requirements for the *Vending Machine* system.

ID	Requirement
1	When the <i>system mode</i> is idle, and the <i>coin sensor</i> changes to true, the coffee machine system shall: reset the <i>request timer</i> , assign choice to the <i>system mode</i> .
2	When the <i>system mode</i> is choice , and the <i>coin sensor</i> is false, and the <i>coin sensor</i> was false, and the <i>coffee request button</i> changes to pressed, and the <i>request timer</i> is lower than or equal to 30.0, the coffee machine system shall: reset the <i>request timer</i> , assign preparing weak coffee to the <i>system mode</i> .

3	When the <i>system mode</i> is choice, and the <i>coin sensor</i> is false, and the <i>coin sensor</i> was false, and the <i>coffee request button</i> changes to pressed, and the <i>request timer</i> is greater than 30.0, the coffee machine system shall: reset the <i>request timer</i> , assign preparing strong coffee to the <i>system mode</i> .
4	When the <i>system mode</i> is preparing weak coffee, and the <i>request timer</i> is greater than or equal to 10.0, and the <i>request timer</i> is lower than or equal to 30.0, the coffee machine system shall: assign idle to the <i>system mode</i> , assign weak to the <i>coffee machine output</i> .
5	When the <i>system mode</i> is preparing strong coffee, and the <i>request timer</i> is greater than or equal to 30.0, and the <i>request timer</i> is lower than or equal to 50.0, the coffee machine system shall : assign idle to the <i>system mode</i> , assign strong to the <i>coffee machine output</i> ..

Table 7.4: List of requirements for the *Vending Machine* system in the SysReq-CNL.

```

1  import static org.junit.Assert.fail;
2
3  import java.io.File;
4  import java.io.IOException;
5  import java.nio.charset.StandardCharsets;
6  import java.nio.file.Files;
7  import java.nio.file.Paths;
8  import java.util.ArrayList;
9  import java.util.List;
10
11 import org.junit.Test;
12
13 public class TestManager_VM {
14
15     private List<Integer> systemMode;
16     private List<Integer> machineOutput;
17     private List<Integer> requestTimer;
18     private List<Integer> time;
19     private List<Boolean> coinSensor;
20     private List<Boolean> coffeRequest;
21
22     @Test
23     public void testAllFiles() throws IOException {
24         File folder = new File("D:\\Git\\gitws\\diss\\tests\\VendingMachine");
25         for (File file : folder.listFiles()) {
26             VendingMachine vm = new VendingMachine();
27             simulate(file, vm);
28         }
29     }
30
31     public void simulate(File csvFile, VendingMachine vm) throws IOException {
32         if (csvFile.getName().equals("mapping.csv"))

```

```

33     return;
34     List<String> lines =
        Files.readAllLines(Paths.get(csvFile.getAbsolutePath()),
            StandardCharsets.UTF_8);
35     int duration = lines.size() - 1;
36     systemMode = new ArrayList<Integer>();
37     machineOutput = new ArrayList<Integer>();
38     requestTimer = new ArrayList<Integer>();
39     coinSensor = new ArrayList<Boolean>();
40     time = new ArrayList<Integer>();
41     coffeRequest = new ArrayList<Boolean>();
42     extractSignals(lines);
43     boolean shouldFail = csvFile.getName().contains("fail") ? true : false;
44     boolean result = true;
45     for (int i = 0; i < duration; i++) {
46         // for debugging purposes
47         if (time.get(i) != i) {
48             fail(i + ": Time was " + time.get(i));
49         }
50         vm.simulate(coffeRequest.get(i), coinSensor.get(i));
51         // if no reset has happened,
52         // increase the request timer by the time that passed during
            simulation, in this case always 1
53         if (!vm.isReset) {
54             vm.the_request_timer = vm.the_request_timer + 1;
55         }
56         vm.old_the_coin_sensor = coinSensor.get(i);
57         vm.old_the_coffee_request_button = coffeRequest.get(i);
58         boolean check = checkOutputs(i, shouldFail, csvFile.getName(), vm);
59         if (!check) {
60             result = false;
61         }
62     }
63
64     if (shouldFail && result) {
65         fail(csvFile.getName() + " passed, but should have failed.");
66     }
67 }
68
69 private void extractSignals(List<String> lines) {
70     String header = lines.get(0);
71     String[] signals = header.split(";");
72     for (int i = 1; i < lines.size(); i++) {
73         String currentLine = lines.get(i);
74         String[] currentLineArray = currentLine.split(";");
75         for (int j = 0; j < currentLineArray.length; j++) {
76             if (signals[j].equals("the coin sensor")) {
77                 coinSensor.add(Integer.parseInt(currentLineArray[j]) == 1 ? true
                    : false);

```



```

78     }
79     if (signals[j].equals("the coffee request sensor")) {
80         coffeRequest.add(Integer.parseInt(currentLineArray[j]) == 1 ?
            true : false);
81     }
82     if (signals[j].equals("the request timer")) {
83         requestTimer.add(Integer.parseInt(currentLineArray[j]));
84     }
85     if (signals[j].equals("the system mode")) {
86         systemMode.add(Integer.parseInt(currentLineArray[j]));
87     }
88     if (signals[j].equals("the coffee machine output")) {
89         machineOutput.add(Integer.parseInt(currentLineArray[j]));
90     }
91     if (signals[j].equals("Time")) {
92         time.add(Integer.parseInt(currentLineArray[j]));
93     }
94 }
95 }
96
97 }
98
99 private boolean checkOutputs(int i, boolean shouldFail, String string,
    VendingMachine vm) {
100     boolean result = vm.the_system_mode == systemMode.get(i);
101     if (!result && !shouldFail) {
102         fail(i + ": Systemmode " + vm.the_system_mode + " but expected " +
            systemMode.get(i) + " in " + string);
103     }
104     boolean result1 = vm.the_coffee_machine_output == machineOutput.get(i);
105     if (!result1 && !shouldFail) {
106         fail(i + ": MachineOutput " + vm.the_coffee_machine_output + " but
            expected " + machineOutput.get(i)
107             + " in " + string);
108     }
109     boolean result2 = vm.the_request_timer == requestTimer.get(i);
110     if (!result2 && !shouldFail) {
111         fail(i + ": RequestTimer " + vm.the_request_timer + " but expected "
            + requestTimer.get(i) + " in "
112             + string);
113     }
114     return (result && result1 && result2);
115 }
116
117 }

```

---

Listing 7.1: Vending Machine Driver for TG-SRL

---

```

1 import static org.junit.Assert.fail;

```

```

2
3 import java.io.File;
4 import java.io.IOException;
5 import java.nio.charset.StandardCharsets;
6 import java.nio.file.Files;
7 import java.nio.file.Paths;
8 import java.util.ArrayList;
9 import java.util.List;
10
11 import org.junit.Test;
12
13 public class TestManager_VM_Nat2Test {
14
15     private List<Integer> systemMode;
16     private List<Integer> machineOutput;
17     private List<Float> time;
18     private List<Boolean> coinSensor;
19     private List<Boolean> coffeRequest;
20
21     @Test
22     public void testAllFiles() throws IOException {
23         File folder = new
24             File("D:\\gitws\\diss\\eval\\VM\\Nat2Test\\TestCases");
25         for (File file : folder.listFiles()) {
26             VendingMachine vm = new VendingMachine();
27             simulate(file, vm);
28         }
29
30     public void simulate(File csvFile, VendingMachine vm) throws IOException {
31         if (csvFile.getName().equals("mapping.csv"))
32             return;
33         List<String> lines =
34             Files.readAllLines(Paths.get(csvFile.getAbsolutePath()),
35                 StandardCharsets.UTF_8);
36         systemMode = new ArrayList<Integer>();
37         machineOutput = new ArrayList<Integer>();
38         coinSensor = new ArrayList<Boolean>();
39         time = new ArrayList<Float>();
40         coffeRequest = new ArrayList<Boolean>();
41         extractSignals(lines);
42         boolean shouldFail = csvFile.getName().contains("fail") ? true : false;
43         boolean result = true;
44         double lastReset = 0;
45         for (int i = 0; i < time.size(); i++) {
46             vm.the_request_timer = time.get(i) - lastReset;
47             vm.simulate(coffeRequest.get(i), coinSensor.get(i));
48             if (vm.isReset) {
49                 lastReset = time.get(i);

```

```

48     }
49     vm.old_the_coin_sensor = coinSensor.get(i);
50     vm.old_the_coffee_request_button = coffeRequest.get(i);
51     boolean check = checkOutputs(i, shouldFail, csvFile.getName(), vm);
52     if (!check) {
53         result = false;
54     }
55 }
56
57 if (shouldFail && result) {
58     fail(csvFile.getName() + " passed, but should have failed.");
59 }
60 }
61
62 private void extractSignals(List<String> lines) {
63     String header = lines.get(0);
64     String[] signals = header.split(";");
65     for (int i = 1; i < lines.size(); i++) {
66         String currentLine = lines.get(i);
67         if (currentLine.isEmpty())
68             continue;
69         String[] currentLineArray = currentLine.split(";");
70         for (int j = 0; j < currentLineArray.length; j++) {
71             if (signals[j].equals("the_coin_sensor")) {
72                 coinSensor.add(Boolean.parseBoolean(currentLineArray[j]));
73             }
74             if (signals[j].equals("the_coffee_request_button")) {
75                 coffeRequest.add(Boolean.parseBoolean(currentLineArray[j]));
76             }
77             if (signals[j].equals("the_system_mode")) {
78                 systemMode.add(Integer.parseInt(currentLineArray[j]));
79             }
80             if (signals[j].equals("the_coffee_machine_output")) {
81                 machineOutput.add(Integer.parseInt(currentLineArray[j]));
82             }
83             if (signals[j].equals("TIME")) {
84                 time.add(Float.parseFloat(currentLineArray[j]));
85             }
86         }
87     }
88 }
89 }
90
91 private boolean checkOutputs(int i, boolean shouldFail, String string,
92     VendingMachine vm) {
93     boolean result = vm.the_system_mode == systemMode.get(i);
94     if (!result && !shouldFail) {
95         fail(i + ": Systemmode " + vm.the_system_mode + " but expected " +
96             systemMode.get(i) + " in " + string);
97     }
98 }

```

```

95     }
96     boolean result1 = vm.the_coffee_machine_output == machineOutput.get(i);
97     if (!result1 && !shouldFail) {
98         fail(i + ": MachineOutput " + vm.the_coffee_machine_output + " but
          expected " + machineOutput.get(i)
99         + " in " + string);
100     }
101     return (result && result1);
102 }
103
104 }

```

Listing 7.2: Vending Machine Driver for Nat2Test

ID	Requirement
1	When the <i>voltage</i> was greater than 80 and the <i>voltage</i> is lower than or equal to 80, the lights controller component shall assign 2 to the <i>indication lights</i> and reset the <i>timer</i> .
2	When the <i>voltage</i> switches to greater than 80 or the <i>mode</i> switched to 1, and the <i>voltage</i> is greater than 80, and the <i>mode</i> was 1, the lights controller component shall assign 1 to the <i>indication lights</i> and reset the <i>timer</i> .
3	When the <i>voltage</i> switches to greater than 80 or the <i>mode</i> switched to 3, and the <i>voltage</i> is greater than 80, and the <i>mode</i> was 3, the lights controller component shall assign 3 to the <i>indication lights</i> and reset the <i>timer</i> .
4	When the <i>voltage</i> switches to greater than 80 or the <i>mode</i> switched to 0, and the <i>voltage</i> is greater than 80, and the <i>mode</i> was 0, the lights controller component shall assign 0 to the <i>indication lights</i> and reset the <i>timer</i> .
5	When the <i>voltage</i> is greater than 80, and the <i>mode</i> was 2, the lights controller component shall assign 2 to the <i>indication lights</i> and reset the <i>timer</i> .
6	When the <i>voltage</i> is greater than 80, and the <i>timer</i> is greater or equal to 34, and the <i>indication lights</i> are 1 or 3, the lights controller component shall assign 2 to the <i>indication lights</i> and reset the <i>timer</i> .
7	When the <i>voltage</i> is greater than 80, and the <i>timer</i> is greater or equal to 22, and the <i>indication lights</i> are 2, and the <i>mode</i> was 1, the lights controller component shall assign 1 to the <i>indication lights</i> and reset the <i>timer</i> .
8	When the <i>voltage</i> is greater than 80, and the <i>timer</i> is greater or equal to 22, and the <i>indication lights</i> are 2, and the <i>mode</i> was 3, the lights controller component shall assign 3 to the <i>indication lights</i> and reset the <i>timer</i> .
9	When the <i>voltage</i> is greater than 80, and the <i>timer</i> is greater or equal to 22, and the <i>indication lights</i> are 2, and the <i>mode</i> was 0, the lights controller component shall assign 0 to the <i>indication lights</i> and reset the <i>timer</i> .
10	When the <i>turn indicator lever</i> switches to 2, and the <i>emergency mode</i> is off, the system shall assign 3 to the <i>mode</i> and reset the <i>timer</i> .
11	When the <i>turn indicator lever</i> switches to 1, and the <i>emergency mode</i> is off, the system shall assign 1 to the <i>mode</i> and reset the <i>timer</i> .

12	When <i>emergency mode</i> switches to true, the system shall set the <i>mode</i> to 0 and reset the <i>timer</i> .
13	When <i>emergency mode</i> is on, and <i>emergency mode</i> was on, and the <i>turn indicator lever</i> switches to 1, the system shall assign 1 to the <i>mode</i> and reset the <i>timer</i> .
14	When <i>emergency mode</i> is on, and <i>emergency mode</i> was on, and the <i>turn indicator lever</i> switches to 2, the system shall assign 3 to the <i>mode</i> and reset the <i>timer</i> .
15	When <i>emergency mode</i> is on, and <i>emergency mode</i> was on, and the <i>turn indicator lever</i> switches to 0, and the <i>mode</i> is not 0, the system shall assign 0 to the <i>mode</i> and reset the <i>timer</i> .
16	When the <i>emergency mode</i> switches to off, and the <i>turn indicator lever</i> is 1, and the <i>turn indicator lever</i> was 1, and the <i>mode</i> is not 1, the system shall assign 1 to the <i>mode</i> and reset the <i>timer</i> .
17	When the <i>emergency mode</i> switches to off, and the <i>turn indicator lever</i> is 2, and the <i>turn indicator lever</i> was 2, and the <i>mode</i> is not 3, the system shall assign 3 to the <i>mode</i> and reset the <i>timer</i> .

Table 7.5: Preprocessed list of requirements for the *Turn Indicator* system.

```

1  public class TIS_Nat2Test {
2
3  // 0: both
4  // 1: left
5  // 2: off
6  // 3: right
7  public int the_indication_lights = 2;
8
9  // 0: both
10 // 1: left
11 // 2: no flashing
12 // 3: right
13 public int the_flashing_mode = 2;
14 public double the_flashing_timer = 0;
15
16 // turn indicator lever:
17 // 0: both/idle
18 // 1: left
19 // 2: right
20 public int old_the_turn_indicator_lever = 0;
21 public boolean old_the_emergency_flashing = false;
22 public int old_the_voltage = 0;
23
24 public int old_the_flashing_mode = 2;
25
26 public boolean isReset = false;
27
28 public void simulate(int the_voltage, int the_turn_indicator_lever,
    boolean the_emergency_flashing) {

```

```

29     int the_flashing_mode_temp = the_flashing_mode;
30     double the_flashing_timer_temp = the_flashing_timer;
31     int the_indication_lights_temp = the_indication_lights;
32     isReset = false;
33     if (((!(the_flashing_mode == 3)) && (old_the_turn_indicator_lever ==
34         2)) && (the_turn_indicator_lever == 2))
35         && (!(old_the_emergency_flashing == false)) &&
36         (the_emergency_flashing == false))) {
37         the_flashing_mode_temp = 3;
38         the_flashing_timer_temp = 0;
39         isReset = true;
40     }
41     if (((the_emergency_flashing == false)
42         && (!(old_the_turn_indicator_lever == 2)) &&
43         (the_turn_indicator_lever == 2)))) {
44         the_flashing_mode_temp = 3;
45         the_flashing_timer_temp = 0;
46         isReset = true;
47     }
48     if (((!(old_the_emergency_flashing)) && (the_emergency_flashing))) {
49         the_flashing_mode_temp = 0;
50         the_flashing_timer_temp = 0;
51         isReset = true;
52     }
53     if (((the_emergency_flashing == false)
54         && (!(old_the_turn_indicator_lever == 1)) &&
55         (the_turn_indicator_lever == 1)))) {
56         the_flashing_mode_temp = 1;
57         the_flashing_timer_temp = 0;
58         isReset = true;
59     }
60     if (((!(old_the_turn_indicator_lever == 2)) &&
61         (the_turn_indicator_lever == 2))
62         && (old_the_emergency_flashing)) && (the_emergency_flashing))) {
63         the_flashing_mode_temp = 3;
64         the_flashing_timer_temp = 0;
65         isReset = true;
66     }
67     if (((!(the_flashing_mode == 0))
68         && (!(old_the_turn_indicator_lever == 0)) &&
69         (the_turn_indicator_lever == 0)))
70         && (old_the_emergency_flashing)) && (the_emergency_flashing))) {
71         the_flashing_mode_temp = 0;
72         the_flashing_timer_temp = 0;
73         isReset = true;
74     }
75     if (((!(old_the_turn_indicator_lever == 1)) &&
76         (the_turn_indicator_lever == 1))
77         && (old_the_emergency_flashing)) && (the_emergency_flashing))) {

```

```

71     the_flashing_mode_temp = 1;
72     the_flashing_timer_temp = 0;
73     isReset = true;
74 }
75 if (((!((the_flashing_mode == 1)) && (old_the_turn_indicator_lever ==
76     1)) && (the_turn_indicator_lever == 1))
77     && (!((old_the_emergency_flashing == false)) &&
78     (the_emergency_flashing == false)))) {
79     the_flashing_mode_temp = 1;
80     the_flashing_timer_temp = 0;
81     isReset = true;
82 }
83 if (!((old_the_voltage <= 80)) && (the_voltage <= 80))) {
84     the_indication_lights_temp = 2;
85     the_flashing_timer_temp = 0;
86     isReset = true;
87 }
88 if (((the_flashing_mode_temp == 1) && (the_voltage > 80))
89     && (!((old_the_flashing_mode == 1)) && (the_flashing_mode_temp ==
90     1))
91     || (!((old_the_voltage > 80)) && (the_voltage > 80)))) {
92     the_indication_lights_temp = 1;
93     the_flashing_timer_temp = 0;
94     isReset = true;
95 }
96 if (((the_flashing_mode_temp == 0) && (the_indication_lights == 2)) &&
97     (the_voltage > 80))
98     && (the_flashing_timer >= 220)) {
99     the_indication_lights_temp = 0;
100     the_flashing_timer_temp = 0;
101     isReset = true;
102 }
103 if (((the_flashing_mode_temp == 1) && (the_indication_lights == 2)) &&
104     (the_voltage > 80))
105     && (the_flashing_timer >= 220)) {
106     the_indication_lights_temp = 1;
107     the_flashing_timer_temp = 0;
108     isReset = true;
109 }
110 if (((the_flashing_mode_temp == 0) && (the_voltage > 80))
111     && (!((old_the_flashing_mode == 0)) && (the_flashing_mode_temp ==
112     0))
113     || (!((old_the_voltage > 80)) && (the_voltage > 80)))) {
114     the_indication_lights_temp = 0;
115     the_flashing_timer_temp = 0;
116     isReset = true;
117 }
118 if (((the_flashing_mode_temp == 2) && (the_voltage > 80))) {
119     the_indication_lights_temp = 2;

```

```

114     the_flashing_timer_temp = 0;
115     isReset = true;
116 }
117 if (((the_indication_lights == 3) || (the_indication_lights == 1)) &&
    (the_voltage > 80))
118     && (the_flashing_timer >= 340)) {
119     the_indication_lights_temp = 2;
120     the_flashing_timer_temp = 0;
121     isReset = true;
122 }
123 if (((the_flashing_mode_temp == 3) && (the_indication_lights == 2)) &&
    (the_voltage > 80))
124     && (the_flashing_timer >= 220)) {
125     the_indication_lights_temp = 3;
126     the_flashing_timer_temp = 0;
127     isReset = true;
128 }
129 if (((the_flashing_mode_temp == 3) && (the_voltage > 80))
130     && (!((old_the_flashing_mode == 3)) && (the_flashing_mode_temp ==
        3))
131     || (!((old_the_voltage > 80)) && (the_voltage > 80)))) {
132     the_indication_lights_temp = 3;
133     the_flashing_timer_temp = 0;
134     isReset = true;
135 }
136 the_flashing_mode = the_flashing_mode_temp;
137 the_flashing_timer = the_flashing_timer_temp;
138 the_indication_lights = the_indication_lights_temp;
139 }
140
141
142 }

```

---

Listing 7.3: Turn Indicator System Implementation for Nat2Test

---

```

1  public class TIS {
2
3      // 0: both
4      // 1: left
5      // 2: off
6      // 3: right
7      public int the_indication_lights = 2;
8
9      // 0: both
10     // 1: left
11     // 2: no flashing
12     // 3: right
13     public int the_flashing_mode = 2;
14     public double the_flashing_timer = 0;

```



```

15
16 // turn indicator lever:
17 // 0: both/idle
18 // 1: left
19 // 2: right
20 public int old_the_turn_indicator_lever = 0;
21 public boolean old_the_emergency_flashing = false;
22 public int old_the_voltage = 0;
23
24 public int old_the_flashing_mode = 2;
25
26 public boolean isReset = false;
27
28 public void simulate(int the_voltage, int the_turn_indicator_lever,
29     boolean the_emergency_flashing) {
30     int the_flashing_mode_temp = the_flashing_mode;
31     double the_flashing_timer_temp = the_flashing_timer;
32     int the_indication_lights_temp = the_indication_lights;
33     isReset = false;
34     if (((!((the_flashing_mode == 3)) && (old_the_turn_indicator_lever ==
35         2)) && (the_turn_indicator_lever == 2))
36         && (!((old_the_emergency_flashing == false)) &&
37             (the_emergency_flashing == false)))) {
38         the_flashing_mode_temp = 3;
39         the_flashing_timer_temp = 0;
40         isReset = true;
41     }
42     if (((the_emergency_flashing == false)
43         && (!((old_the_turn_indicator_lever == 2)) &&
44             (the_turn_indicator_lever == 2)))) {
45         the_flashing_mode_temp = 3;
46         the_flashing_timer_temp = 0;
47         isReset = true;
48     }
49     if (((!((old_the_emergency_flashing)) && (the_emergency_flashing))) {
50         the_flashing_mode_temp = 0;
51         the_flashing_timer_temp = 0;
52         isReset = true;
53     }
54     if (((the_emergency_flashing == false)
55         && (!((old_the_turn_indicator_lever == 1)) &&
56             (the_turn_indicator_lever == 1)))) {
57         the_flashing_mode_temp = 1;
58         the_flashing_timer_temp = 0;
59         isReset = true;
60     }
61     if (((!((old_the_turn_indicator_lever == 2)) &&
62         (the_turn_indicator_lever == 2))
63         && (old_the_emergency_flashing)) && (the_emergency_flashing))) {

```

```

58     the_flashing_mode_temp = 3;
59     the_flashing_timer_temp = 0;
60     isReset = true;
61 }
62 if (((!((the_flashing_mode == 0))
63     && (!((old_the_turn_indicator_lever == 0)) &&
64         (the_turn_indicator_lever == 0)))
65     && (old_the_emergency_flashing)) && (the_emergency_flashing))) {
66     the_flashing_mode_temp = 0;
67     the_flashing_timer_temp = 0;
68     isReset = true;
69 }
70 if (((!((old_the_turn_indicator_lever == 1)) &&
71     (the_turn_indicator_lever == 1))
72     && (old_the_emergency_flashing)) && (the_emergency_flashing))) {
73     the_flashing_mode_temp = 1;
74     the_flashing_timer_temp = 0;
75     isReset = true;
76 }
77 if (((!((the_flashing_mode == 1)) && (old_the_turn_indicator_lever ==
78     1)) && (the_turn_indicator_lever == 1))
79     && (!((old_the_emergency_flashing == false)) &&
80         (the_emergency_flashing == false)))) {
81     the_flashing_mode_temp = 1;
82     the_flashing_timer_temp = 0;
83     isReset = true;
84 }
85 if (((!((old_the_voltage <= 80)) && (the_voltage <= 80))) {
86     the_indication_lights_temp = 2;
87     the_flashing_timer_temp = 0;
88     isReset = true;
89 }
90 if (((the_flashing_mode == 1) && (the_voltage > 80))
91     && (!((old_the_flashing_mode == 1)) && (the_flashing_mode == 1))
92     || (!((old_the_voltage > 80)) && (the_voltage > 80)))) {
93     the_indication_lights_temp = 1;
94     the_flashing_timer_temp = 0;
95     isReset = true;
96 }
97 if (((the_flashing_mode == 0) && (the_indication_lights == 2)) &&
98     (the_voltage > 80))
99     && (the_flashing_timer >= 22)) {
100     the_indication_lights_temp = 0;
101     the_flashing_timer_temp = 0;
102     isReset = true;
103 }
104 if (((the_flashing_mode == 1) && (the_indication_lights == 2)) &&
105     (the_voltage > 80))
106     && (the_flashing_timer >= 22)) {

```

```

101     the_indication_lights_temp = 1;
102     the_flashing_timer_temp = 0;
103     isReset = true;
104 }
105 if (((the_flashing_mode == 0) && (the_voltage > 80))
106     && (!((old_the_flashing_mode == 0)) && (the_flashing_mode == 0))
107     || (!((old_the_voltage > 80)) && (the_voltage > 80)))) {
108     the_indication_lights_temp = 0;
109     the_flashing_timer_temp = 0;
110     isReset = true;
111 }
112 if (((the_flashing_mode == 2) && (the_voltage > 80))) {
113     the_indication_lights_temp = 2;
114     the_flashing_timer_temp = 0;
115     isReset = true;
116 }
117 if (((the_indication_lights == 3) || (the_indication_lights == 1)) &&
118     (the_voltage > 80))
119     && (the_flashing_timer >= 34)) {
120     the_indication_lights_temp = 2;
121     the_flashing_timer_temp = 0;
122     isReset = true;
123 }
124 if (((the_flashing_mode == 3) && (the_indication_lights == 2)) &&
125     (the_voltage > 80))
126     && (the_flashing_timer >= 22)) {
127     the_indication_lights_temp = 3;
128     the_flashing_timer_temp = 0;
129     isReset = true;
130 }
131 if (((the_flashing_mode == 3) && (the_voltage > 80))
132     && (!((old_the_flashing_mode == 3)) && (the_flashing_mode == 3))
133     || (!((old_the_voltage > 80)) && (the_voltage > 80)))) {
134     the_indication_lights_temp = 3;
135     the_flashing_timer_temp = 0;
136     isReset = true;
137 }
138 the_flashing_mode = the_flashing_mode_temp;
139 the_flashing_timer = the_flashing_timer_temp;
140 the_indication_lights = the_indication_lights_temp;
141 }

```

---

Listing 7.4: Turn Indicator System Implementation for TG-SRL

---

```

1
2 import static org.junit.Assert.fail;
3

```

```

4  import java.io.File;
5  import java.io.IOException;
6  import java.nio.charset.StandardCharsets;
7  import java.nio.file.Files;
8  import java.nio.file.Paths;
9  import java.util.ArrayList;
10 import java.util.List;
11
12 import org.junit.Test;
13
14 public class TestManager_TIS {
15
16     private List<Integer> indicationLights;
17     private List<Integer> flashingMode;
18     private List<Integer> flashingTimer;
19     private List<Integer> time;
20     private List<Integer> voltage;
21     private List<Boolean> emergencyFlashing;
22     private List<Integer> turnLever;
23
24     @Test
25     public void testAllFiles() throws IOException {
26         File folder = new File("D:\\Git\\gitws\\diss\\tests\\TIS");
27
28         for (File file : folder.listFiles()) {
29             TIS system = new TIS();
30             simulate(file, system);
31         }
32     }
33
34     public void simulate(File csvFile, TIS vm) throws IOException {
35         if (csvFile.getName().equals("mapping.csv"))
36             return;
37         List<String> lines =
38             Files.readAllLines(Paths.get(csvFile.getAbsolutePath()),
39                 StandardCharsets.UTF_8);
40         int duration = lines.size() - 1;
41         indicationLights = new ArrayList<Integer>();
42         flashingMode = new ArrayList<Integer>();
43         flashingTimer = new ArrayList<Integer>();
44         voltage = new ArrayList<Integer>();
45         time = new ArrayList<Integer>();
46         emergencyFlashing = new ArrayList<Boolean>();
47         turnLever = new ArrayList<Integer>();
48         extractSignals(lines);
49         boolean shouldFail = csvFile.getName().contains("fail") ? true :
50             false;
51         boolean result = true;
52         for (int i = 0; i < duration; i++) {

```

```

50     if (time.get(i) != i) {
51         fail(i + ": Time was " + time.get(i));
52     }
53     vm.simulate(voltage.get(i), turnLever.get(i),
54                 emergencyFlashing.get(i));
55     // if no reset has happened,
56     // increase the flashing timer by the time that passed during
57     // simulation, in this case always 1
58     if (!vm.isReset) {
59         vm.the_flashing_timer = vm.the_flashing_timer + 1;
60     }
61     vm.old_the_emergency_flashing = emergencyFlashing.get(i);
62     // because it is an output, the "old" variable refers to i-1
63     if (i > 0)
64         vm.old_the_flashing_mode = flashingMode.get(i - 1);
65     else
66         vm.old_the_flashing_mode = 2;
67     vm.old_the_voltage = voltage.get(i);
68     vm.old_the_turn_indicator_lever = turnLever.get(i);
69     boolean check = checkOutputs(i, shouldFail, csvFile.getName(), vm);
70     if (!check) {
71         result = false;
72     }
73 }
74
75 if (shouldFail && result) {
76     fail(csvFile.getName() + " passed, but should have failed.");
77 }
78
79 private void extractSignals(List<String> lines) {
80     String header = lines.get(0);
81     String[] signals = header.split(";");
82     for (int i = 1; i < lines.size(); i++) {
83         String currentLine = lines.get(i);
84         String[] currentLineArray = currentLine.split(";");
85         for (int j = 0; j < currentLineArray.length; j++) {
86             if (signals[j].equals("the voltage")) {
87                 voltage.add(Integer.parseInt(currentLineArray[j]));
88             }
89             if (signals[j].equals("emergency mode")) {
90                 emergencyFlashing.add(Integer.parseInt(currentLineArray[j]) ==
91                                     1 ? true : false);
92             }
93             if (signals[j].equals("the timer")) {
94                 flashingTimer.add(Integer.parseInt(currentLineArray[j]));
95             }
96             if (signals[j].equals("the indication lights")) {

```

```

96         indicationLights.add(Integer.parseInt(currentLineArray[j]));
97     }
98     if (signals[j].equals("the mode")) {
99         flashingMode.add(Integer.parseInt(currentLineArray[j]));
100    }
101    if (signals[j].equals("the turn indicator lever")) {
102        turnLever.add(Integer.parseInt(currentLineArray[j]));
103    }
104    if (signals[j].equals("Time")) {
105        time.add(Integer.parseInt(currentLineArray[j]));
106    }
107    }
108    }
109
110    }
111
112    private boolean checkOutputs(int i, boolean shouldFail, String string,
113                                TIS vm) {
114        boolean result = vm.the_indication_lights == indicationLights.get(i);
115        if (!result && !shouldFail) {
116            fail(i + ": IndicationLights " + vm.the_indication_lights + " but
117                expected " + indicationLights.get(i)
118                + " in " + string);
119        }
120        boolean result1 = vm.the_flashing_mode == flashingMode.get(i);
121        if (!result1 && !shouldFail) {
122            fail(i + ": FlashingMode " + vm.the_flashing_mode + " but expected
123                " + flashingMode.get(i) + " in "
124                + string);
125        }
126        boolean result2 = vm.the_flashing_timer == flashingTimer.get(i);
127        if (!result2 && !shouldFail) {
128            fail(i + ": FlashingTimer " + vm.the_flashing_timer + " but
129                expected " + flashingTimer.get(i) + " in "
130                + string);
131        }
132        return (result && result1 && result2);
133    }
134    }
135    }

```

---

Listing 7.5: Turn Indicator System Driver for TG-SRL

---

```

1
2 import static org.junit.Assert.fail;
3
4 import java.io.File;
5 import java.io.IOException;
6 import java.nio.charset.StandardCharsets;

```

```

7  import java.nio.file.Files;
8  import java.nio.file.Paths;
9  import java.util.ArrayList;
10 import java.util.List;
11
12 import org.junit.Test;
13
14 public class TestManager_TIS_Nat2Test {
15
16     private List<Integer> indicationLights;
17     private List<Integer> flashingMode;
18     private List<Float> time;
19     private List<Integer> voltage;
20     private List<Boolean> emergencyFlashing;
21     private List<Integer> turnLever;
22
23     @Test
24     public void testAllFiles() throws IOException {
25         File folder = new File("D:\\gitws\\diss\\files\\eval\\TI\\TestCases");
26
27         for (File file : folder.listFiles()) {
28             TIS_Nat2Test vm = new TIS_Nat2Test();
29             simulate(file, vm);
30         }
31     }
32
33     public void simulate(File csvFile, TIS_Nat2Test vm) throws IOException {
34         if (csvFile.getName().equals("mapping.csv"))
35             return;
36         List<String> lines =
37             Files.readAllLines(Paths.get(csvFile.getAbsolutePath()),
38                 StandardCharsets.UTF_8);
39         indicationLights = new ArrayList<Integer>();
40         flashingMode = new ArrayList<Integer>();
41         voltage = new ArrayList<Integer>();
42         time = new ArrayList<Float>();
43         emergencyFlashing = new ArrayList<Boolean>();
44         turnLever = new ArrayList<Integer>();
45         extractSignals(lines);
46         boolean shouldFail = csvFile.getName().contains("fail") ? true : false;
47         boolean result = true;
48         double lastReset = 0;
49         for (int i = 0; i < time.size(); i++) {
50             vm.the_flashing_timer = time.get(i) - lastReset;
51             vm.simulate(voltage.get(i), turnLever.get(i),
52                 emergencyFlashing.get(i));
53             if (vm.isReset) {
54                 lastReset = time.get(i);
55             }
56         }
57     }
58 }

```

```

53     }
54     vm.old_the_emergency_flashing = emergencyFlashing.get(i);
55     // because it is an output, the "old" variable refers to i-1
56     if (i > 0)
57         vm.old_the_flashing_mode = flashingMode.get(i - 1);
58     else
59         vm.old_the_flashing_mode = 2;
60     vm.old_the_voltage = voltage.get(i);
61     vm.old_the_turn_indicator_lever = turnLever.get(i);
62     boolean check = checkOutputs(i, shouldFail, csvFile.getName(), vm);
63     if (!check) {
64         result = false;
65     }
66 }
67
68 if (shouldFail && result) {
69     fail(csvFile.getName() + " passed, but should have failed.");
70 }
71 }
72
73 private void extractSignals(List<String> lines) {
74     String header = lines.get(0);
75     String[] signals = header.split(";");
76     for (int i = 1; i < lines.size(); i++) {
77         String currentLine = lines.get(i);
78         if (currentLine.isEmpty())
79             continue;
80         String[] currentLineArray = currentLine.split(";");
81         for (int j = 0; j < currentLineArray.length; j++) {
82             if (signals[j].equals("the_voltage")) {
83                 voltage.add(Integer.parseInt(currentLineArray[j]));
84             }
85             if (signals[j].equals("the_emergency_flashing")) {
86                 emergencyFlashing.add(Integer.parseInt(currentLineArray[j]) == 1
87                     ? true : false);
88             }
89             if (signals[j].equals("the_indication_lights")) {
90                 indicationLights.add(Integer.parseInt(currentLineArray[j]));
91             }
92             if (signals[j].equals("the_flashing_mode")) {
93                 flashingMode.add(Integer.parseInt(currentLineArray[j]));
94             }
95             if (signals[j].equals("the_turn_indicator_lever")) {
96                 turnLever.add(Integer.parseInt(currentLineArray[j]));
97             }
98             if (signals[j].equals("TIME")) {
99                 time.add(Float.parseFloat(currentLineArray[j]));
100             }
101         }
102     }

```



```

101     }
102
103 }
104
105 private boolean checkOutputs(int i, boolean shouldFail, String string,
106     TIS_Nat2Test vm) {
107     boolean result = vm.the_indication_lights == indicationLights.get(i);
108     if (!result && !shouldFail) {
109         fail(i + ": IndicationLights " + vm.the_indication_lights + " but
110             expected " + indicationLights.get(i)
111             + " in " + string);
112     }
113     boolean result1 = vm.the_flashing_mode == flashingMode.get(i);
114     if (!result1 && !shouldFail) {
115         fail(i + ": FlashingMode " + vm.the_flashing_mode + " but expected "
116             + flashingMode.get(i) + " in "
117             + string);
118     }
119     return (result && result1);
120 }
121 }

```

Listing 7.6: Turn Indicator System Driver for Nat2Test

Time	the coffee machine output	the coffee request sensor	the coin sensor	the request timer	the system mode
0	0	0	0	1	1
1	0	0	1	0	0
2	0	0	0	1	0
3	0	1	0	0	3
4	0	1	0	1	3
5	0	0	0	2	3
6	0	0	0	3	3
7	0	0	0	4	3
8	0	0	0	5	3
9	0	0	0	6	3
10	0	0	0	7	3
11	0	0	0	8	3
12	0	0	0	9	3
13	0	0	0	10	3
14	1	0	0	11	1
15	1	0	1	0	0
16	1	0	1	1	0

17	1	0	0	2	0
18	1	0	0	3	0
19	1	0	0	4	0
20	1	0	0	5	0
21	1	0	0	6	0
22	1	0	0	7	0
23	1	0	0	8	0
24	1	0	0	9	0
25	1	0	0	10	0
26	1	0	0	11	0
27	1	0	0	12	0
28	1	0	0	13	0
29	1	0	0	14	0
30	1	0	0	15	0
31	1	0	0	16	0
32	1	0	0	17	0
33	1	0	0	18	0
34	1	0	0	19	0
35	1	0	0	20	0
36	1	0	0	21	0
37	1	0	0	22	0
38	1	0	0	23	0
39	1	0	0	24	0
40	1	0	0	25	0
41	1	0	0	26	0
42	1	0	0	27	0
43	1	0	0	28	0
44	1	0	1	29	0
45	1	1	0	30	0
46	1	0	0	31	0
47	1	1	0	0	2
48	1	1	0	1	2
49	1	0	0	2	2
50	1	1	0	3	2
51	1	0	0	4	2
52	1	1	0	5	2
53	1	0	0	6	2

---

Table 7.7: Exemplary test case from TG-SRL for the *Vending Machine* system.

---

```

1  public class VendingMachine {
2
3  // mode
4  // 0: choice, 1: idle, 2: preparing strong coffee, 3: preparing weak
   coffee
5  public int the_system_mode = 1;
6  public int the_coffee_machine_output = 0;
7  public double the_request_timer = 0;
8  public boolean isReset = false;
9
10 public boolean old_the_coin_sensor = false;
11 public boolean old_the_coffee_request_button = false;
12
13 public void simulate(boolean the_coffee_request_button, boolean
   the_coin_sensor ) {
14     // Req1
15     if ((the_request_timer <= 30.0) && (the_request_timer >= 10.0) &&
        (the_system_mode == 3)) {
16         the_coffee_machine_output = 1;
17         the_system_mode = 1;
18         isReset = false;
19     // Req2
20     } else if ((the_request_timer <= 50.0) && (the_request_timer >= 30.0)
        && (the_system_mode == 2)) {
21         the_coffee_machine_output = 0;
22         the_system_mode = 1;
23         isReset = false;
24     // Req3
25     } else if ((the_coin_sensor == true && !(old_the_coin_sensor ==
        true)) && (the_system_mode == 1)) {
26         the_request_timer = 0;
27         isReset = true;
28         the_system_mode = 0;
29     // Req4
30     } else if ((the_coffee_request_button == true &&
        !(old_the_coffee_request_button == true))
31         && (old_the_coin_sensor == false) && (the_coin_sensor == false)
        && (the_request_timer <= 30.0)
32         && (the_system_mode == 0)) {
33         the_request_timer = 0;
34         isReset = true;
35         the_system_mode = 3;
36     // Req5
37     } else if ((the_coffee_request_button == true &&
        !(old_the_coffee_request_button == true))
38         && (old_the_coin_sensor == false) && (the_coin_sensor == false)
        && (the_request_timer > 30.0)
39         && (the_system_mode == 0)) {
40         the_request_timer = 0;
41         isReset = true;
42         the_system_mode = 2;
43     } else {
44         isReset = false;
45     }
46 }
47 }

```

---

Frame Element	Description
#43	Thing (coin)
#45	Thing (is)
#47	Predicate (inserted), ARG1: #43 (coin), ARGM-TMP: #48 (while the machine is in state 1 and the request timer is lower or equal to 30)
#49	Thing (machine)
#50	Thing (state 1)
#55	Cardinal (1)
#57	Predicate (request), ARG1: #58 (timer)
#58	Thing (timer)
#61	Predicate (is lower), ARG1: #58 (timer), ARG2: #62 (or equal to 30)
#62	Thing (30)
#65	Cardinal (30)
#67	Thing (then)
#69	Thing (machine state)
#71	Thing (should)
#73	Predicate (be modified), ARGM-ADV: #67 (then), ARG1: #69 (machine state), ARGM-MOD: #71 (should), ARG2: #74 (3)
#74	Cardinal (3)
#77	Predicate (request)
#79	Thing (timer)
#81	Thing (should)
#83	Predicate (be reset), ARG1: #79 (timer), ARGM-MOD: #81 (should), ARG2: #84 (reset)
#84	Thing (reset)

Table 7.6: Output from [46] on the sentence "If a coin is inserted while the machine is in state 1 and the request timer is lower or equal to 30, then the machine state should be modified to 3 and the request timer should be reset."

## Bibliography

- [1] Umar Zakir Abdul Hamid. *Autonomous, Connected, Electric and Shared Vehicles: Disrupting the Automotive and Mobility Sectors*. SAE International, 2022.
- [2] International Organization for Standardization. ISO 26262:2018: Road vehicles -Functional Safety, 2018. <https://www.iso.org/standard/68383.html>.
- [3] Mich Luisa, Franch Mariangela, and Novi Inverardi Pierluigi. Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9:40–56, 2004.
- [4] Felix Beringhoff, Joel Greenyer, Christian Roesener, and Matthias Tichy. Thirty-one challenges in testing automated vehicles: Interviews with experts from industry and research. In *2022 IEEE Intelligent Vehicles Symposium (IV)*, pages 360–366. IEEE, 2022.
- [5] VDA Working Group 13. Automotive SPICE Process Assessment / Reference Model, 2023. Version 4.0, <https://vda-qmc.de/automotive-spice/>.
- [6] Anurag Dwarakanath and Shubhashis Sengupta. Litmus: Generation of test cases from functional requirements in natural language. In *Natural Language Processing and Information Systems: 17th International Conference on Applications of Natural Language to Information Systems*, pages 58–69. Springer, 2012.
- [7] Colin J. Neill and Phillip A. Laplante. Requirements engineering: The state of the practice. *IEEE software*, 20(6):40–45, 2003.
- [8] Gustavo Carvalho, Diogo Falcão, Flávia Barros, Augusto Sampaio, Alexandre Mota, Leonardo Motta, and Mark Blackburn. NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications. *Science of Computer Programming*, 95:275–297, 2014.
- [9] Alexander Kugler. TG-SRL GitHub Repository. <https://git.rwth-aachen.de/informatik11/test-generation-with-srl>.
- [10] Norman Hansen, Norbert Wiechowski, Alexander Kugler, Stefan Kowalewski, Thomas Rambow, and Rainer Busch. Model-in-the-Loop and Software-in-the-Loop Testing of Closed-Loop Automotive Software with Arttest. pages 1537–1549. Gesellschaft für Informatik, Bonn, 2017.
- [11] Norbert Wiechowski, Alexander Kugler, Norman Hansen, Stefan Kowalewski, Thomas Rambow, and Rainer Busch. Arttest - A New Test Environment for

- Model-Based Software Development. Technical report, SAE Technical Paper, 2017. <https://doi.org/10.4271/2017-01-0004>.
- [12] OMG Systems Modeling Language (SysML). SysML Website. <https://sysml.org/>, accessed: 2023-06-17.
- [13] AUTomotive Open System ARchitecture (AUTOSAR). *AUTOSAR Classic Platform*. <https://www.autosar.org/standards/classic-platform/>, accessed: 2023-05-22.
- [14] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.
- [15] Glenford J. Myers, Tom Badgett, Todd M. Thomas, and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2011.
- [16] Christopher Kugler. *Systematic Derivation of Feature-Driven and Risk-Based Test Strategies for Automotive Applications*. PhD thesis, Informatik 11 - Embedded Software, RWTH Aachen University, 2023. <https://publications.rwth-aachen.de/record/959607>.
- [17] Gustavo Carvalho, Flávia Barros, Florian Lapschies, Uwe Schulze, and Jan Peleska. Model-based testing from controlled natural language requirements. In *Formal Techniques for Safety-Critical Systems: Second International Workshop, FTSCS 2013, Queenstown, New Zealand, October 29–30, 2013. Revised Selected Papers 2*, pages 19–35. Springer, 2014.
- [18] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. *Attempto Controlled English for Knowledge Representation*, pages 104–124. Springer Berlin Heidelberg, 2008.
- [19] Aerospace and Defence Industries Association of Europe. *ASD-STE100: Simplified Technical English*. ASD, 2013. <https://asd-ste100.org/>.
- [20] Rolf Schwitter. Specifying weak constraints in Processable English. In *Proceedings of the Australasian Computer Science Week Multiconference*, pages 1–4, 2019.
- [21] Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.
- [22] Peter Clark, William R Murray, Phil Harrison, and John Thompson. Naturalness vs. predictability: A key debate in controlled languages. In *International Workshop on Controlled Natural Language*, pages 65–81. Springer, 2009.
- [23] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, et al. Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):1–76, 2009.

- [24] Sidney Nogueira, Hugo Araujo, Renata Araujo, Juliano Iyoda, and Augusto Sampaio. Test case generation, selection and coverage from natural language. *Science of Computer Programming*, 181:84–110, 2019.
- [25] Charles Antony Richard Hoare et al. *Communicating Sequential Processes*, volume 178. Prentice-hall Englewood Cliffs, 1985.
- [26] Dennis Tsichritzis. The equivalence problem of simple programs. *Journal of the ACM (JACM)*, 17(4):729–738, 1970.
- [27] Christopher Manning and Hinrich Schutze. *Foundations of statistical natural language processing*. MIT press, 1999.
- [28] Adwait Ratnaparkhi. A linear observed time statistical parser based on maximum entropy models. *arXiv preprint cmp-lg/9706014*, 1997.
- [29] Michael Collins. Head-Driven Statistical Models for Natural Language Parsing. *Computational Linguistics*, 29(4):589–637, 2003.
- [30] Michael Collins and Terry Koo. Discriminative Reranking for Natural Language Parsing. *Computational Linguistics*, 31(1):25–70, 2005.
- [31] Srinivas Bangalore and Aravind Joshi. Supertagging: An approach to almost parsing. *Computational linguistics*, 25(2):237–265, 1999.
- [32] Eugene Charniak. A Maximum-Entropy-Inspired Parser. In *1st Meeting of the North American Chapter of the Association for Computational Linguistics*, 2000.
- [33] Mitch Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational linguistics*, 19(2):313–330, 1993.
- [34] Anne Abeillé. *Treebanks: Building and using parsed corpora*, volume 20. Springer Science & Business Media, 2003.
- [35] Martha Palmer, Daniel Gildea, and Paul Kingsbury. The proposition bank: An annotated corpus of semantic roles. *Computational linguistics*, 31(1):71–106, 2005.
- [36] Yoav Goldberg. *Neural network methods for natural language processing*. Springer Nature, 2022.
- [37] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- [38] Prakash M. Nadkarni, Lucila Ohno-Machado, and Wendy W. Chapman. Natural language processing: An introduction. *Journal of the American Medical Informatics Association*, 18(5):544–551, 2011.

- [39] Meishan Zhang. A survey of syntactic-semantic parsing based on constituent and dependency structures. *Science China Technological Sciences*, 63(10):1898–1920, 2020.
- [40] George A. Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 38(11):39–41, 1995.
- [41] Allen Institute for AI. AllenNLP Semantic Role Labeling Demo, 2023. <https://demo.allennlp.org/semantic-role-labeling>, accessed: 2023-05-18.
- [42] Naveen Arivazhagan, Christos Christodoulopoulos, and Dan Roth. Labeling the semantic roles of commas. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.
- [43] International Computer Science Institute (ICSI). FrameNet, 2023. <https://framenet.icsi.berkeley.edu/>, accessed: 2023-05-18.
- [44] Swabha Swayamdipta, Sam Thomson, Chris Dyer, and Noah A. Smith. Frame-Semantic Parsing with Softmax-Margin Segmental RNNs and a Syntactic Scaffold. *arXiv preprint arXiv:1706.09528*, 2017. <https://github.com/swabhs/open-sesame>, accessed: 2023-05-18.
- [45] Daniel Khashabi, Mark Sammons, Ben Zhou, Tom Redman, Christos Christodoulopoulos, Vivek Srikumar, Nick Rizzolo, Lev Ratinov, Guanheng Luo, Quang Do, et al. CogCompNLP: Your swiss army knife for NLP. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.
- [46] Michael Ringgaard, Rahul Gupta, and Fernando CN Pereira. SLING: A framework for frame semantic parsing. *arXiv preprint arXiv:1710.07032*, 2017. <https://github.com/ringgaard/sling>, accessed: 2023-05-18.
- [47] Dipanjan Das, Desai Chen, André FT Martins, Nathan Schneider, and Noah A. Smith. Frame-semantic parsing. *Computational linguistics*, 40(1):9–56, 2014.
- [48] Conference on Computational Natural Language Learning (CoNLL) Website. <https://www.conll.org/>, accessed: 2023-05-18.
- [49] Xavier Carreras and Lluís Màrquez. Introduction to the CoNLL-2005 Shared Task: Semantic Role Labeling. In *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL-2005)*, pages 152–164. Association for Computational Linguistics, 2005.
- [50] Sameer S. Pradhan and Nianwen Xue. OntoNotes: the 90% solution. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, pages 11–12, 2009.



- [51] OntoNotes Release 5.0. <https://catalog.ldc.upenn.edu/LDC2013T19>, 2013. accessed: 2023-05-18.
- [52] Collin F. Baker, Michael Ellsworth, and Katrin Erk. SemEval-2007 task 19: Frame semantic structure extraction. In *Proceedings of the Fourth International Workshop on Semantic Evaluations (SemEval-2007)*, pages 99–104, 2007.
- [53] Peng Shi and Jimmy Lin. Simple BERT Models for Relation Extraction and Semantic Role Labeling. *arXiv preprint arXiv:1904.05255*, 2019.
- [54] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving Language Understanding by Generative Pre-Training. 2018. OpenAI. <https://www.mikecaptain.com/resources/pdf/GPT-1.pdf>.
- [55] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep Contextualized Word Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237. Association for Computational Linguistics, 2018.
- [56] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A Survey on Model-Based Testing Approaches: A Systematic Review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pages 31–36, 2007.
- [57] Rolf Schwitter and Norbert E. Fuchs. Attempto - from specifications in controlled natural language towards executable specifications. *arXiv preprint cmp-lg/9603004*, 1996.
- [58] Stephen G. MacDonell, Kyongho Min, and Andy M. Connor. Autonomous requirements specification processing using natural language processing. *arXiv preprint arXiv:1407.6099*, 2014.
- [59] Jaroslav Drazan and Vladimir Mencl. Improved processing of textual use cases: Deriving behavior specifications. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 856–868. Springer, 2007.
- [60] Valdivino Alexandre De Santiago Junior and Nandamudi Lankalapalli Vijaykumar. Generating model-based test cases from natural language requirements for space application software. *Software Quality Journal*, 20:77–143, 2012.
- [61] Ravishankar Boddu, Lan Guo, Supratik Mukhopadhyay, and Bojan Cukic. RETNA: from requirements to testing in a natural way. In *Proceedings. 12th IEEE International Requirements Engineering Conference*, pages 262–271. IEEE, 2004.

- [62] Mitch Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. The Penn Treebank: Annotating predicate argument structure. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey*, 1994.
- [63] Patrick Blackburn, Johan Bos, Michael Kohlhase, and Hans De Nivelle. Inference and computational semantics. pages 11–28, 2001.
- [64] Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel Briand, and Zohaib Iqbal. Automatic generation of system test cases from use case specifications. In *Proceedings of the 2015 international symposium on software testing and analysis*, pages 385–396, 2015.
- [65] Tao Yue, Lionel C. Briand, and Yvan Labiche. An automated approach to transform use cases into activity diagrams. In *Modelling Foundations and Applications: 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings 6*, pages 337–353. Springer, 2010.
- [66] Avik Sinha, Stanley M. Sutton, and Amit Paradkar. Text2Test: Automated inspection of natural language use cases. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 155–164. IEEE, 2010.
- [67] Avik Sinha, Amit Paradkar, Palani Kumanan, and Branimir Boguraev. A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 327–336. IEEE, 2009.
- [68] Matthias Schnelte. Generating test cases for timed systems from controlled natural language specifications. In *2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, pages 348–353. IEEE, 2009.
- [69] Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.
- [70] Gustavo Carvalho, Flávia Barros, Ana Carvalho, Ana Cavalcanti, Alexandre Mota, and Augusto Sampaio. NAT2TEST tool: From natural language requirements to test cases based on CSP. In *Software Engineering and Formal Methods: 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*, pages 283–290. Springer, 2015.
- [71] Bruno Oliveira, Gustavo Carvalho, Mohammad Reza Mousavi, and Augusto Sampaio. Simulation of hybrid systems from natural-language requirements. In *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*, pages 1320–1325. IEEE, 2017.
- [72] Elham S. Khorasani. Artificial intelligence: Structures and strategies for complex problem solving. *Scalable Computing: Practice and Experience*, 9(3), 2008.

- [73] Karin Kipper Schuler. *VerbNet: A broad-coverage, comprehensive verb lexicon*. University of Pennsylvania, 2005.
- [74] Pradhan Sameer, Skatje Myers, and Tim Gorman. PropBank Frame Files. GitHub repository: <https://github.com/propbank/propbank-frames/>.
- [75] The University of Colorado Boulder. PropBank Frameset for "to add". <https://verbs.colorado.edu/propbank/framesets-english-aliases/add.html>, accessed: 2023-06-09.
- [76] Cognitive Computation Group. Semantic Role Labeling Demo, 2023. [https://cogcomp.seas.upenn.edu/page/demo\\_view/SRLEnglish](https://cogcomp.seas.upenn.edu/page/demo_view/SRLEnglish), accessed: 2023-05-18.
- [77] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.
- [78] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [79] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. GitHub repository: <https://github.com/Z3Prover/z3>.
- [80] Ayal Klein, Jonathan Mamou, Valentina Pyatkin, Daniela Stepanov, Hangfeng He, Dan Roth, Luke Zettlemoyer, and Ido Dagan. QANom: Question-answer driven SRL for nominalizations. In *Proceedings of the 28th international conference on computational linguistics*, pages 3069–3083, 2020.
- [81] Vivek Srikumar and Dan Roth. Modeling semantic relations expressed by prepositions. *Transactions of the Association for Computational Linguistics*, 1:231–242, 2013.
- [82] Dipanjan Das, Desai Chen, Andre F. T. Martins, Nathan Schneider, and Noah A. Smith. SEMAFOR Website. <http://www.cs.cmu.edu/~ark/SEMAFOR/>, accessed: 2023-06-17.
- [83] MathWorks. MATLAB and Simulink Website. <https://www.mathworks.com/products/matlab.html>, accessed: 2023-05-22.
- [84] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005. <http://www.cs.gmu.edu/~offutt/mujava/>, accessed: 2023-05-22.
- [85] Yu-Seung Ma and Jeff Offutt. Description of class mutation operators for Java. *Electronics and Telecommunications Research Institute, Korea*, 2005. [https://www.researchgate.net/publication/228965184\\_Description\\_of\\_class\\_mutation\\_mutation\\_operators\\_for\\_java](https://www.researchgate.net/publication/228965184_Description_of_class_mutation_mutation_operators_for_java).

## Bibliography

- [86] Nat2Test Website. <https://www.cin.ufpe.br/~ghpc/nat2test/>. accessed: 2023-05-22.
- [87] MathWorks. Simulink Variant Subsystems Website. <https://de.mathworks.com/help/simulink/ug/variant-subsystems.html>, accessed: 2023-05-22.
- [88] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*, 2023.
- [89] OpenAI. ChatGPT Chat Link. <https://chatgpt.com/share/6796c95d-37c0-8006-9a17-2deedf269314> accessed: 2025-01-19.