Mälardalen University Doctoral Dissertation 336

Industrial System Level Test Automation

Daniel Flemström

MÄLARDALEN UNIVERSITY
SWEDEN

Mälardalen University Press Dissertations
No. 336

INDUSTRIAL SYSTEM LEVEL TEST AUTOMATION

Daniel Flemström

2021

School of Innovation, Design and Engineering

Copyright © Daniel Flemström, 2021
ISBN 978-91-7485-522-7
ISSN 1651-4238
Printed by E-Print AB, Stockholm, Sweden

Mälardalen University Press Dissertations
No. 336

INDUSTRIAL SYSTEM LEVEL TEST AUTOMATION

Daniel Flemström

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid Akademin för innovation, design och teknik kommer att offentligen försvaras fredagen den 22 oktober 2021, 13.15 i Delta & online, Mälardalens högskola, Västerås.

Fakultetsopponent: Professor Andy Zaidman, Delft Institute of Technology

Akademin för innovation, design och teknik

Abstract

Vehicular software systems control and monitor many safety-critical functions, such as automated emergency brakes and anti-spin. These functions are integrated and tested at system level to ensure that the entire system works as intended.Traditionally, these functions or selected combinations of functions are tested in isolation. Although desired, rigorous testing of combinations of functions prior to deployment is seldom possible. One reason is the overwhelming work required to write new test cases for the nearly infinite combinations of functions and driving scenarios. Since software testing already accounts for up to 60% of the software development cost and the execution time in test rigs is expensive, further testing must be achieved by running test cases more in parallel. Moreover, new test cases must be reusable in different driving scenarios to cover more combinations of functions. One solution is to express the test logic in a new way so that it can be executed in parallel, independently of other tests and independent of the input stimuli. This would allow reusing the test logic for different sequences or drive scenarios. Passive testing is one such approach that has not yet been used much for vehicular software due to perceived difficulties, although it is well-established in other domains. One particular problem is how to specify and execute passive test cases for vehicular systems in an, for the test engineer, intuitive and straightforward manner. Further, there is a lack of tool support and knowledge on efficiently applying passive testing in an industrial context. Thus, the overall research goal of this thesis is to propose and evaluate industrially applicable methods and tools for passive testing at the system level of vehicular software systems. The research is based on a series of papers and case studies within the vehicular industry. In contrast to existing specification languages based on formal mathematical expressions, considerable effort was spent creating an intuitive language and an interactive tool, simple but powerful enough, to encourage the industrial application of passive testing. The main contributions include an easy-to-write and easy-to-read description language for passive test cases, an interactive development environment, and knowledge on how to succeed in passive testing in an industrial software development process. The findings of this thesis contribute to making passive testing a viable method for system-level testing and a way of reusing test logic. In the case of a studied train control management system, using passive testing may reuse up to 50% of the test logic for the safety-related requirements and 10% of the non-safety-related requirements.

ISBN 978-91-7485-522-7
ISSN 1651-4238

To my family

If something's hard to do, then it's not worth doing

Homer Simpson

Sammanfattning

Datorsystemen i våra fordon styr och övervakar många säkerhetskritiska funktioner, såsom automatiserade nödbromsar och antispinn. Dessa funktioner är integrerade och testade på systemnivå för att säkerställa att hela systemet fungerar som avsett. Traditionellt testas dessa funktioner eller utvalda kombinationer av funktioner var för sig. Detta tillvägagångssätt kräver många testfall som ofta är utförda efter varandra, med liten eller ingen variation. Eftersom programvarutestning redan står för upp till 60% av kostnaden för utveckling av programvara, och exekveringstiden i testriggar är dyr, måste ytterligare testning uppnås genom att köra testfall mer parallellt. Vidare måste nya testfall vara återanvändbara i olika körscenarier. Dessa måste täcka fler kombinationer av funktioner utan att testlogiken, som kontrollerar att kraven uppfylls, behöver dupliceras eller skrivas om. En lösning är att göra testlogiken oberoende av test stimuli, så att testlogiken kan köras parallellt och återanvändas för olika stimulisekvenser eller scenarier. Passiv testning är en sådan metod som ännu inte har använts mycket för fordonsprogramvara, även om den är väl etablerad inom andra områden. De problem som behöver lösas är t.ex. hur man specificerar och utvärderar passiva testfall för fordonssystem på ett intuitivt och enkelt sätt. Vidare behövs verktygsstöd och kunskap om hur man effektivt tillämpar passiv testning i ett industriellt sammanhang. Därför är den här avhandlingens övergripande mål att föreslå och utvärdera industriellt tillämpbara metoder och verktyg för passiv testning på systemnivå för fordonsprogramvarusystem. Avhandlingen baseras på en serie artiklar och fallstudier genomförda inom fordonsindustrin. Huvudbidragen inkluderar, ett för testingenjören, intuitivt beskrivningsspråk för passiva testfall som är lätt att skriva och läsa, en interaktiv utvecklingsmiljö för passiva testfall och kunskap om hur man lyckas med passiv testning i en industriell mjukvaruutvecklingsprocess. Resultaten bidrar till att göra passiv testning attraktivt och lättillgängligt för testning på systemnivå samt att främjar återanvändning av testlogik. Ett konkret exempel på detta är ett kontrollsystem för tåg som studerades. Här skulle passiv testning kunna bidra till att återanvända upp till 50% av testlogiken för de säkerhetskritiska kraven och upp till 10% av de icke-säkerhetskritiska kraven.

Abstract

Vehicular software systems control and monitor many safety-critical functions, such as automated emergency brakes and anti-spin. These functions are integrated and tested at system level to ensure that the entire system works as intended. Traditionally, these functions or selected combinations of functions are tested in isolation. Although desired, rigorous testing of combinations of functions prior to deployment is seldom possible. One reason is the overwhelming work required to write new test cases for the nearly infinite combinations of functions and driving scenarios. Since software testing already accounts for up to 60 % of the software development cost and the execution time in test rigs is expensive, further testing must be achieved by running test cases more in parallel. Moreover, new test cases must be reusable in different driving scenarios to cover more combinations of functions. One solution is to express the test logic in a new way so that it can be executed in parallel, independently of other tests and independent of the input stimuli. This would allow reusing the test logic for different sequences or drive scenarios. Passive testing is one such approach that has not yet been used much for vehicular software due to perceived difficulties, although it is well-established in other domains. One particular problem is how to specify and execute passive test cases for vehicular systems in an intuitive and straightforward manner. Further, there is a lack of tool support and knowledge on efficiently applying passive testing in an industrial context. Thus, the overall research goal of this thesis is to propose and evaluate industrially applicable methods and tools for passive testing at the system level of vehicular software systems. The research is based on a series of papers and case studies within the vehicular industry. In contrast to existing specification languages based on formal mathematical expressions, considerable effort was spent creating an intuitive language and an interactive tool, simple but powerful enough, to encourage the industrial application of passive testing. The main contributions include an easy-to-write and easy-to-read description language for passive test cases, an interactive development environment, and knowledge on how to succeed in passive testing in an industrial software development process. The findings of this thesis contribute to making passive testing a viable method for system-level testing and a way of reusing test logic. In the case of a studied train control management system, using passive testing may reuse up to 50% of the test logic for the safety-related requirements and 10% of the non-safety-related requirements.

Acknowledgments

Many people have supported me on this long and sometimes winding journey. Above all, I thank my family that has put up with my mental and physical absence these years. I have also been very fortunate to have a team of supervisors curling me beyond expectations: Prof. Wasif Afzal, Dr. Eduard Enoiu, and Prof. Daniel Sundmark. Thank you for pushing me forward when my inner doubts threatened to take over. Without you, this path would have been much duller and at least a decade longer. I also have been very fortunate to have such a supportive and knowledgeable primary industrial contact as Ola Sellin.
Of course, I would not forget to mention the people at Alstom (former Bombardier Transportation Sweden AB) in Västerås, RISE, Mälardalen University, Volvo Construction Equipment and ABB HVDC that have contributed with their kind support and fruitful discussions.

Finally, I would like to thank the Swedish Governmental Agency of Innovation (Vinnova), Mälardalen University, Volvo Construction Equipment, Alstom (former Bombardier Transportation Sweden AB), and ABB HVDC. Your support has made this thesis possible via IMPRINT, TESTMINE, ADEPTNESS and the XIVT projects.

Västerås, October 2021

List of Publications

Publications Included in the Thesis[1][2]

This thesis is based on the following studies:

Paper A: *Similarity-Based Prioritization of Test Case Automation.* **Daniel Flemström**, Pasqualina Potena, Daniel Sundmark, Wasif Afzal, Markus Bohlin. Published in the Software Quality Journal, Special Issue on Automation of Software Test: Improving Practical Applicability (SQJ'2018).

Paper B: *From Natural Language Requirements to Passive Test Cases using Guarded Assertions.* **Daniel Flemström**, Eduard Enoiu, Wasif Azal, Daniel Sundmark, Thomas Gustafsson, Avenir Kobetski. Published at the International Conference on Software Quality, Reliability and Security (QRS'18).

Paper C: *A Case Study of Interactive Development of Passive Tests.* **Daniel Flemström**, Thomas Gustafsson, Avenir Kobetski. Published at the International Workshop on Requirements Engineering and Testing (RET'18).

Paper D: *Specification of Passive Test Cases using an Improved T-EARS Language.* **Daniel Flemström**, Wasif Afzal, Eduard Paul Enoiu. Accepted to the International Conference on Software Quality (SWQD'22).

Paper E: *Industrial-Scale Passive Testing with T-EARS.* **Daniel Flemström**, Henrik Jonsson, Eduard Enoiu, Wasif Azal Published at the International Conference on Software Testing, Verification and Validation (ICST'21) (Best Paper Award).

Statement of Contribution

In the included publications the first author was the primary driver and contributor to the research, approach and tool development, study design, data collection, analysis and reporting of the research work. However, the initial prototype of T-EARS and the tool in Paper B and C was a joint effort by all the authors.

[1]The included publications are reformatted to comply with the thesis printing format.

[2]All published studies included in this thesis are reprinted with explicit permission from the copyright holders (i.e. IEEE and Springer).

Additional Publications not Included in the Thesis

1. *Vertical Test Reuse for Embedded Systems: A Systematic Mapping Study.* **Daniel Flemström**, Daniel Sundmark and Wasif Afzal. Published at The 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA'15).

2. *Similarity Function Evaluation.* **Daniel Flemström**. Published at Mälardalen Real TIme Research Centre, Mälardalen University), Västerås, Sweden.

3. *Exploring Test Overlap in System Integration: An Industrial Case Study.* **Daniel Flemström**, Wasif Afzal, Daniel Sundmark. Published at The 42nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA'16).

4. *A Research Roadmap for Test Design in Automated Integration Testing of Vehicular Systems.* **Daniel Flemström**, Thomas Gustafsson, Avenir Kobetski, Daniel Sundmark. Published at The Second International Conference on Fundamentals and Advances in Software Systems Integration (FASSI'16).

5. *SAGA Toolbox: Interactive Testing of Guarded Assertions.* **Daniel Flemström**,Thomas Gustafsson, Avenir Kobetski. Published at the IEEE International Conference on Software Testing, Verification and Validation (ICST'17).

6. *Improving Introductory Programming Courses by Using a Simple Accelerated Graphics Library.* Thomas Larsson, **Daniel Flemström**. Published at The Annual SIGRAD Conference; Special Theme: Computer Graphics in Healthcare (SIGRAD'07).

7. *A Prototype Tool for Software Component Services in Embedded Real-time Systems.* Frank Lüders, **Daniel Flemström**, Anders Wall and Ivica Crnkovic. Published at The International Symposium on Component-Based Software Engineering (CBSE'06).

8. *Software Components Services for Embedded Real-Time.* Frank Lüders, **Daniel Flemström**/ and Anders Wall. Published at The Working IEEE/IFIP Conference on Software Architecture (WICSA'05).

Contents

Sammanfattning v

Abstract vii

Acknowledgments vii

List of Publications xi

I Thesis Summary 1

1 Introduction 3

2 Background 5
- 1 Software Testing 5
- 2 System Level Testing 7
- 3 Requirements, Test Cases And Test Automation 10
- 4 Passive Testing 11

3 Research Summary 15
- 1 Research Goals 15
- 2 Scope and Limitations 16
- 3 Empirical Research in an Industrial Context 16
  - 3.1 Challenges in Empirical Research 16
  - 3.2 The Technology Transfer Model 18

4 Research Contributions 21
- 1 RG1 - Methods and Tools For Passive Testing 23
  - 1.1 C1: An Intuitive and Easy To Use Specification Language . . 23
  - 1.2 C2: An Interactive Passive Test Case Development Environment 25
  - 1.3 C3: Research Framework 33
  - 1.4 C4: Way of Working 37
- 2 RG2 - Evaluation 43

5 Related Work 47

6 Conclusions 49

7 Future Work 51

Bibliography 52

II Papers **57**

A Similarity-Based Prioritization of Test Case Automation 59
1 Introduction 61
2 Background 63
2.1 Previous Work 63
2.2 Preliminaries 65
2.3 Problem Statement 66
3 Reducing Manual Test Execution Effort Using Similarity-based Reuse and Automation Order Prioritization. 67
3.1 Assumptions 67
3.2 Test Case Automation 68
3.3 Measuring Manual Execution Effort 70
3.4 Comparing Approaches 71
3.5 An Example of Similarity-Based Reuse 72
3.6 Formal Problem Definition 74
3.7 Potential/Effort **(P/E)** Prioritization 74
4 Industrial Case Study 76
4.1 Case Study Context 77
4.2 Units of Analysis 77
4.3 Variables and Measurements 79
4.4 Assumptions and Simplifications 79
4.5 Experimental Settings 79
4.6 Execution and Data Collection Procedures 81
5 Industrial Case Study Results 81
5.1 Execution Performance of the Prioritization Algorithms 87
6 Threats to Validity 89
7 Conclusions and Future Work 89
References 90

B From Natural Language Requirements to Passive Test Cases using Guarded Assertions 95
1 INTRODUCTION 97
2 Background 98
3 Guarded Assertions, T-EARS and the SAGA Tool Chain 99
3.1 Concepts and Principles 100
3.2 Event-driven Guarded Assertion 101
3.3 State-driven Guarded Assertion 102
3.4 Guarded Assertions for Ubiquitous Requirements 103
4 Translating Requirements into T-EARS Guarded Assertions 104

4.1 Requirements Analysis 106
4.2 Abstract G/A Construction 107
4.3 Implementation Analysis 110
4.4 G/A Concretization 111
4.5 Tuning and Validation 111
5 Proof of Concept Evaluation 111
5.1 Requirements Analysis 112
5.2 Abstract G/A Construction 113
5.3 Implementation Analysis 117
5.4 G/A Concretization 117
5.5 Tuning and Validation 118
6 Related work 119
7 Discussion 120
7.1 Towards Industrial Adoption of SAGA and G/As 120
7.2 Limitations 120
8 Conclusion and Future Work 120
References 121

C A Case Study of Interactive Development of Passive Tests 125
1 Introduction 127
2 Background 128
2.1 Guarded Assertions 128
2.2 Timed Easy Approach to Requirements Syntax (T-EARS) . . 129
2.3 Toolbox 129
3 Case Study Design 130
3.1 Objective and Method Selection 130
3.2 Context 130
3.3 Preparation and Data Collection 131
3.4 Analysis Procedures 133
3.5 Validity 135
4 Case Study Results 136
4.1 GA Approach 136
4.2 Language 138
4.3 Tool 141
5 Related Work 143
6 Conclusions and Future Work 143
References 144

D Specification of Passive Test Cases using an Improved T-EARS Language 147
1 Introduction 149
2 Background 150
2.1 Passive Testing 150
2.2 Guarded Assertions 151
2.3 Easy Approach to Requirements Syntax (EARS) 151

2.4 The Ohm Grammar Language 151
3 Method 152
4 Result: The Updated T-EARS Language 152
4.1 Keyword Terminals 155
4.2 Structural Elements 156
4.3 Basic Data Types 156
4.4 Signals Data Type 157
4.5 Intervals Data Type 158
4.6 Events Data Type 160
4.7 Boolean Expressions 161
4.8 Guarded Assertion Rules 161
4.9 Miscellaneous Modifiers 162
4.10 Timing Considerations 162
4.11 General Structure of a T-EARS Test Case 163
5 Related Work 164
6 Discussion on T-EARS Improvement 164
7 Conclusion and Future Work 166
References 166

E Industrial-Scale Passive Testing with T-EARS 171
1 Introduction 173
2 Background to T-EARS and its Tool chain 175
3 Method 177
3.1 Study Objective 177
3.2 Case organization & Unit of Analysis 177
3.3 Safety Related Requirements 178
3.4 Case Study Procedure 179
4 Results and Discussion 180
4.1 Phase I - Gold Standard and Requirements Selection 181
4.2 Phase II - Requirement Analysis Results 181
4.3 Phase II - Abstract G/A Construction Results 182
4.4 Phase II - Implementation Analysis Results 183
4.5 Phase II - Concretization Results 185
4.6 Phase II - Tuning and Validation Results 185
4.7 Phase III - Final Evaluation 191
5 Related Work 192
6 Conclusions and Future Work 193
References 194

Part I

Thesis Summary

Chapter 1

Introduction

From being purely mechanical constructs, vehicles now have an increasing number of functions that are controlled by software. The execution of that software is divided and distributed on numerous onboard computers, and the software eventually controls physical entities, e.g., brakes or steering. Thus, the software in a vehicle is a large and complex software system that must be tested meticulously. However, the increasing number of possible function combinations is challenging to test, and even when the testing is fully automated, there are numerous problems such as repetitive testing and wasted testing resources [18]. One suggested way to improve such testing is to use passive testing approaches [6, 8]. The idea is to continuously observe the system and test each requirement whenever it makes sense. Although some initial work [21, 31] has proven the approach viable for the vehicular industry, passive testing has met some resistance from practitioners [3, 10, 14] due to its formal approach to specify the test cases. While current approaches to reduce such resistance include pre-defined patterns [3, 14] and graphical representations [13] to facilitate the formalization of either requirements or test cases, this thesis follows the path of [17, 21, 31], focussing on creating a more industrially acceptable, highly intuitive description language together with an interactive development environment for passive test cases. The overall research goal of this thesis is to propose and evaluate industrially applicable methods and tools for passive testing at the system level of vehicular software systems. The main contributions of this thesis are:

(C1) - An intuitive and easy to use specification language (T-EARS),

(C2) - An interactive integrated development and analysis environment (Napkin),

(C3) - A modular open source research platform for passive testing tools,

(C4) - A structured process for translating natural language requirements to passive test cases (Way of Working), and

(C5) - Industrial evaluations of the proposed solutions (Industrial Evaluations).

The thesis is organized into two parts: "Thesis Summary" and "Papers'. The thesis summary aims to outline the preliminaries and the research goals and ultimately combine and present the main findings at a higher level. In contrast, specific details of the results and research methods are found in the "Papers" part in the included published papers.

Chapter 2

Background

This section outlines the central concepts and principles used in this thesis. First out is an overview of software testing's purpose and concepts in Section 1. The overview is followed by further details on applicable concepts and principles at one particular level of software testing: system level testing in Section 2. Section 3 introduces concepts such as requirements, test cases and test automation. Finally, Section 4 presents a discussion on passive testing.

1 Software Testing

Intuitively, the purpose of software testing is to make sure that the developed system works as expected [1]. However, the focus of software testing has shifted over time, from being a debugging aid to a structured approach for revealing bugs.

Today, there are many different testing approaches for different kinds of software systems [30]. In this thesis, the focus is on vehicular *embedded* system software. Embedded systems typically interact with hardware that controls physical entities, such as actuators and sensors [5], as opposed to desktop applications such as Microsoft Word™, that mainly interacts with a user. Another difference is that the addressed systems are signal-based. While a C++ or Java program communicates using function calls with complex data as arguments between different system modules, signal-based systems have an interface that resembles an electrical wire diagram. Each wire, or signal, represents a value over time of a simple type (e.g., Integer, Float, or Binary). Furthermore, the flow of such programs is more deterministic than a C++ program: All program modules are collected into tasks that executes at a fixed cycle time, e.g., each 250ms. At the beginning of each execution cycle, all input signals of a task are read. Then each module calculates its resulting output. Finally, all out-signals of the task are written and thus available to other tasks. This principle of signal-based communication is maintained at all levels of abstraction, although several signals are in practice packed together and transported as data bus telegrams at the system level. When testing or debugging such a system, the signals are recorded at various sample rates. A set of test cases can then observe and compare the input and output signals to determine whether the system fulfills its requirements or not.

The safety critical nature of vehicular software requires a well-documented software

development process. The workflow of such software development processes are often described using the V-model [19]. Since its introduction in the 90s, the V-model has been widely accepted in the embedded software community, and it has been adopted in numerous variants [5, 22]. This section uses a simplified version of the V-model to illustrate the concepts and principles used in this thesis. Figure 2.1 illustrates the relevant concepts of that model. For a more extensive introduction to the V-model, the reader is referred to the work of Forsberg et al. [19].

At the left side of the V-model, the system is broken down into several abstraction levels that correspond to an increasing level of detail in the specification. The top-level specifies the complete product with software and hardware, while the bottom level specifies the software implementation units. These units are successively integrated and tested on the V-model's right side to form the complete product finally.

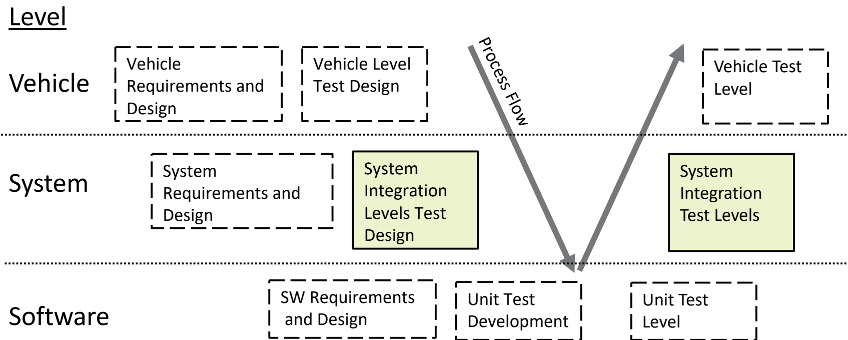


Figure 2.1: V-Model Adapted to Illustrate the Studied Projects. The model shows requirements and test specifications created at the left side of the V. Integration and testing takes place at the right side.

Looking into more detail at the V-model, the functional specifications and their corresponding test specifications are developed iteratively at each level. The functional specifications are passed on to the next level of abstraction as soon as they have been approved. As the level of detail increases, more details are added to the design. At the bottom of the V-model, the specifications have been divided into implementable units that can be tested in (more or less) isolation. As soon as units are implemented and tested, the integration work begins. The integration work is illustrated in Figure 2.1 by the right process flow arrow in the V-model. At the integration level(s), the implemented units are integrated into larger subsystems until the integrated subsystems, together with the target hardware, form a complete product. A set of test cases with level-specific test objectives are created at each such level. Examples of such objectives are code coverage (at the unit level) and interface compliance or timing accuracy at the system integration levels [30]. These differences are further discussed in Section 2.

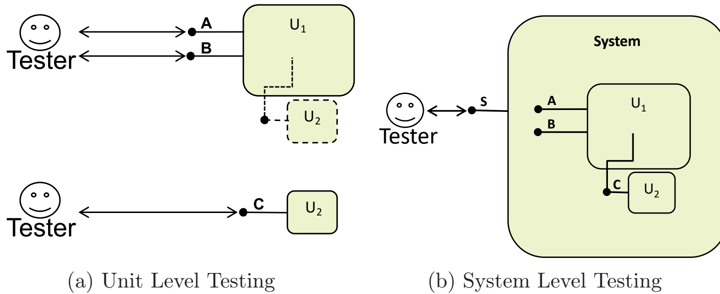
2 System Level Testing

As previously discussed in Section 1, the purpose of the right hand side of the V-model is to assemble all software units into a resulting, complete software system.

Depending on the system's characteristics under development, a different number of *system integration levels* and *integration strategies* may be applied. Pezzè and Young [30] suggest an integration level where the test objective is based on software module compatibility, followed by a system-level that explicitly targets the end-to-end functionality, also referred to as end-to-end testing [36]. Major differences between these levels, as observed during the work with this thesis, include the input/output space, and the communication format. The input space is a lot bigger at the system level since there are tens of thousands of signals with values that vary over time. A unit typically has a handful signals as input and output. Further, the different functions constituting a system may or may not interfere with each other. Another word for this is feature interaction [7] that can be intended or unintended. One important job of the system level tester is to find unknown or unintended feature interactions that some time cause the system to diverge from its specification. The other observed difference between the different levels of abstraction is the means of communication. This poses some challenges when logging signals, since it requires different logger hardware that may have clocks that are slightly out of synch. This is especially evident when parts of the system is simulated in a PC, while the tested software artifacts execute in a cycle correct real-time context. The PC has one time domain (wall clock time), while the simulator may need several seconds to simulate one second. Even though the software attempts to compensate, the experiences from Papers B, C and E show that the difference is still substantial and non deterministic. Finally, the sheer amount of available signals at the system level is a challenge on its own. Oftentimes there are limits on the number of signals that can be logged.

Pezzè and Young [30] describe different integration strategies, where the simplest strategy is the "no strategy" or big-bang integration. This means that no integration work is done until all software has been implemented. As a consequence, all modules are integrated and tested at the same time. Other strategies such as "bottom up" or "top down" integration, promotes successive integration of the system in small increments with thorough testing at each increment. Such a successive approach requires that the parts of the system that have not yet been implemented must be stubbed or simulated. The difference between a stub and a simulated subsystem is that the former returns some default value for each request, regardless of the input values, while the latter returns some estimated value based on the input. Sometimes a combination of the bottom up and top down approach is required (Pezzè and Young define this as the sandwich approach in [30]).

Figure 2.2 illustrates the motivation behind a successive integration strategy. In particular, it shows the difference between unit-level tests and system level tests. Figure 2.2a shows that a unit-level approach allows the tester to verify the units U_1 and U_2 independently, keeping the focus of the tests on the functional and non-functional behavior of the tested unit. Any faults are most probably confined to either U_1 or U_2, narrowing the search scope. Further, as illustrated in the topmost



(a) Unit Level Testing (b) System Level Testing

Figure 2.2: Testing at Different Levels of Abstraction

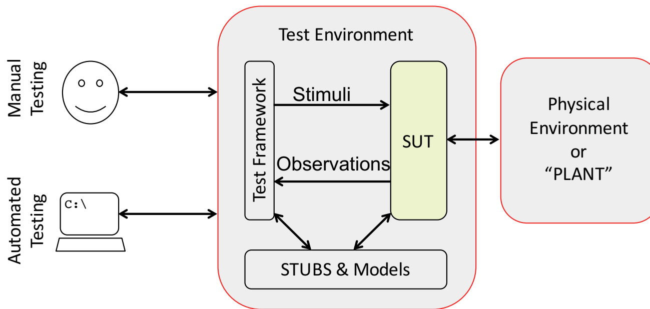


Figure 2.3: Conceptual Overview of a Test Setup. The System Under Test (SUT) is driven by a test framework and interacting with a physical environment using stubs and simulation for the parts of the system that are not available.

unit U_1, any dependencies to other units (U_2 in the figure) are stubbed or simulated. This is indicated by the dashed line around U_2 in Figure 2.2a.

As units get ready, they are passed on to the integration level. Although a fault may origin from inside *either* U_1 or U_2, the tester assumes that the units work according to their specification and focus the testing on, e.g., interactions between units. When the system is complete, as illustrated in Figure 2.2b, focus is moved to testing of end-to-end behavior (System Level Testing). Note that, in Figure 2.2b, the level of abstraction only allows access to the interface (**S**), since lower-level interfaces such as **A**, **B**, **C** may be hidden. For larger systems, one integration level may not be sufficient. Instead the system is built from levels of successively larger sub-systems until the whole system is complete. The number of abstraction layers may also vary from project to project. A project that concerns software that works closely to hardware, may include an additional hardware-software integration level. Figure 2.3 illustrates an embedded system test environment adapted from the work of Broekman [5]. The most central part in the figure is the *SUT* which is an abbreviation for **S**ystem **U**nder **T**est. The SUT is the part(s) of the system that is currently subject to testing. Any other parts of the system, such as sub-systems not yet implemented, are often implemented as *stubs*. A stub is a minimal implementation of an interface (e.g., returning hardcoded default values). The SUT in the Figure

interacts with a *physical environment* (sometimes also called the “Plant”), consisting of actuators and sensors. This environment is typically realized partly by hardware and partly by real-time simulation models. If the testing objective is functional verification, i.e., switching on a lamp in a train cabin, a purely simulated environment may be sufficient. On the other hand, if the objective of the test activity is to verify that the timing behavior is according to specification, a software simulated approach is often not adequate. In such cases, the software needs to be executed on the target hardware, while most of the remaining environment may be realized as a real-time simulation model.

Figure 2.3 also illustrates a generic test environment, controlled and managed by a *Test Framework*. A test framework provides means of managing and operating the system under test, either for *Manual Testing* or *Automated Testing*. Typical tasks for such a framework are handling configuration data and providing a programmable interface to facilitate scripting stimuli to the SUT.

3 Requirements, Test Cases And Test Automation

At any level of abstraction in the V-model, there is a set of **requirements** that describe the expectations of the system at that level. These expectations may concern aspects like functional behavior or non-functional behavior such as timing or robustness. There are many ways to express such requirements. A few examples are natural language (NL), natural language with some restrictions such as REG [3] and EARS [26]. There are also numerous model-based approaches such as UML[1], SYSML[2] etc. In this thesis, the focus is on near-natural language requirements. The projects at Alstom Transport AB in Västerås studied in this thesis concern the Train Control and Management Software(TCMS) for different trains. These projects used natural language for the *regular* (not safety-critical) requirements and a semi-formal natural language for the *safe*(safety-critical) requirements. The semi-formal requirements start with a natural language description, followed by a list of inputs and the expected outputs, expressed as abstract signal expressions. According to our industrial partner, this semi-formal expression of requirements is aligned with the railway safety standards [11, 9, 12] for safety-critical functions.

Given a set of requirements, the test team must figure out how to show that the tested item works according to the specified requirements. This can be done by defining a set of plausible scenarios, or situations, that reflect realistic use of the tested item. The standard ISO 29119-1[23] defines a scenario as: “A scenario can be a user story, use-case, operational concept, or sequence of events the software may encounter etc.”. Especially in safety-critical parts of the software, the scenario keeps track of the requirements it coverers. Often several scenarios or situations are required to show that a particular requirement is met. Each scenario then results in a set of test cases. ISO 29119-1[23] defines a test case as “A test case is a set of test case preconditions, inputs (including actions, where applicable), and expected results, developed to drive the execution of a test item to meet test objectives ...”. These actions and expected results vary, depending on what the covered part of the system is. The test item (or test object) can be a unit or a complete system. Furthermore, the test objectives describe what to focus on. For example, at the unit testing level, the unit (e.g., a function) is the test object, a test objective may concern the correctness of a calculation made by the test object while the test goal can be a coverage criterion such as covering each line of code. During the system integration level(s), the test object is a combination of two or more subsystems, and the test objective concerns the interaction between the (sub) systems or how well the overall integrated system functions. The test goal may be a coverage criterion, such as covering all interfaces or combinations of subsystems. At the system level, the test object is a complete (sub) system, and a test objective may concern the end-to-end timing or result of a system request. The test goal can be to cover a set of system requirements in a number of critical situations or scenarios.

As stated in the ISO 29119-1 definition of a (traditional) test case, the instructions for checking that the reaction is correct (also called the test logic) would be intertwined

[1]https://www.uml.org/
[2]https://sysml.org/

with the stimulus instructions. An advantage is that since test case execution always starts from a known system state, the current system state is known at each step since the tester knows what he has done in the previous steps. One of the drawbacks is that the automated code for the test logic must be repeated if tested in different scenarios. A concrete example from the train-software TCMS studied in Paper E is that the safety-critical part had 116 requirements tested at 207 places in different test cases. A great deal of the effort could be saved if the test logic could be written once and reused for the remaining 91 occurrences. This indicates that it would be beneficial to separate the test logic from the stimulus sequence. The possibility of such separation and thus reusing the test logic is one of the expected benefits of passive testing, presented in the next section.

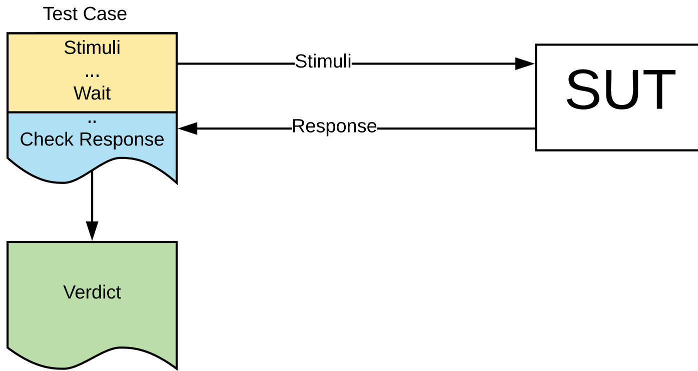
4 Passive Testing

While traditional test cases, like the ones in Section 3, contain test steps including *both* stimuli to the system under test and instructions for evaluating the system's response, passive testing focuses entirely on observations of the tested system and compares this with the expected response. This separation of concerns may facilitate reuse of the test logic for many different stimuli sequences. Further, such passive test cases are typically automated, and since passive test cases do not interfere with the state of the system at all, they can execute in parallel and independently of each other. One consequence is that the tester (or a test sequence generator) may focus on creating realistic input sequences that put the system under test in as many testable states as possible.

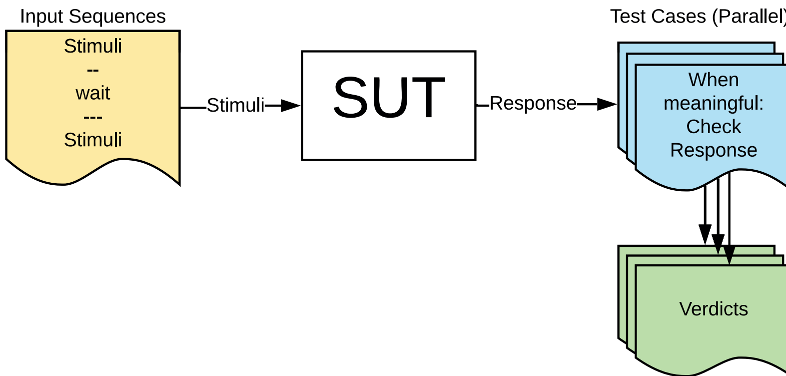
Even though passive test cases do not *alter* the system state, they are highly *dependent* on it. A passive test needs to "know" when it is meaningful to perform a test and when it is not. E.g., there is no point in checking a requirement concerning top speed, when the vehicle is not even started.

When a tester writes a test case for active testing, he/she has full control over the system state. This means that the current system state can be inferred since it always starts from a known starting point followed by a known sequence of test stimuli. In contrast, when writing a test case for passive testing, the test case needs to infer the current state of the system entirely by reading trace output of the system. There is no known restart points as the test case is evaluated over the complete execution.

Figure 2.4 illustrates the in-practice difference between the two approaches. Beginning with the case of active testing, Figure 2.4a shows an example of how a test case is executed. Before executing each such test case, the tester or automation framework would reset the system under test. As described in Section 3, a test case in active testing consists of a series of test steps. Each test step describes a stimulus to send to the system under test, and an (optional) expected response to the given stimulus. Executing a test step thus means that the stimulus is sent to the system under test, either manually by a tester or using a test automation framework as described in Section 2. To be sure that the system has had time to respond, testers tend to wait for a long period to make sure that the system has got time to respond



(a) Active Test Cases Contain Intertwined Stimulus and Test Logic



(b) Passive Test Cases Only Contain the Test Logic

Figure 2.4: Sequential Active vs Parallel Passive Testing

as expected. The system is then probed and compared to the expected response. Depending on how well the response matches the expected response, the test cases are noted as “passed”, or “failed”. If a test step fails, the whole test case is noted as failed.

Some shortcomings of the active approach include:

- Restarting the system under test between each test is time consuming,
- a requirement is only tested at one point in time, and
- time may be wasted due to the long to-be-sure timeouts.

Turning to the case of passive testing, the stimulus has been separated from the actual testing, as illustrated in Figure 2.4b. This separation of concerns opens up some attractive possibilities: As shown in the figure, all test cases can be executed in parallel. The tester may now create or generate stimuli for different purposes without the need of rewriting the test logic in the test case. Further, passive testing contributes to mitigate the shortcomings mentioned above as:

- Restarting the system under test between each test is no longer necessary, and
- a requirement is tested as many times as the system enters the corresponding testable state,
- Instead of time-outs, as soon as the system enters a testable state, the corresponding test case can be evaluated to pass or fail directly.

A more in depth general review of passive testing methods can be found in [8]. While those methods primarily target test protocols and web services, another approach to passive testing has shown promising results for system-level testing of vehicular systems. The approach is called *Independent Guarded Assertions* [21, 31] or G/A for short. A G/A is a test case that only observes the system under test. It consists of a guard and its assertions. The *guard* part describes under which circumstances a set of *assertions* should hold. The assertion part carries the test logic of a requirement that should be met.

Consider the example of a light that should always be lit as long as a push-button is pressed. The test logic, evaluating whether the light is lit or not, corresponds to the assertion part. The part ensuring that the assertion is only considered whenever the button is pressed corresponds to the guard part.

Describing such guards and assertions is challenging since they contain logical expressions (maybe the engine needs to be on, AND the button is pressed) and temporal constraints or expectations (how long is a button press and how long is it acceptable to wait until the lamp is lit?). Although there are several ways of expressing temporal logic, such as timed automata using UPPAAL as in [31] or by more mathematical means as in [8] , the test engineers at the system level participating in the studies behind this thesis requested more intuitive and straightforward means of expressing passive test cases. Finding ways of expressing passive test cases thus plays a central role in this thesis.

Finally, the terms Passive Test Case, Independent Guarded Assertion, and Guarded Assertion (G/A for short) are used interchangeably throughout the thesis.

Chapter 3

Research Summary

1 Research Goals

Software testing, and especially vehicular system testing at an industrial scale, faces many challenges. One such challenge is testing a large set of safety-critical functions that are supposed to work under a large number of scenarios and situations. Passive testing has been suggested to meet these challenges by, e.g., allowing reuse of test logic between situations. Although successfully used in other areas, industrial usage in the vehicular domain has been hindered by perceived difficulties due to the mathematical formalisms traditionally used for expressing the test cases and the lack of efficient ways of working with passive testing.

The overall research goal is to propose and evaluate industrially applicable methods and tools for passive testing at the system level of vehicular software systems.

The overall research goal can be sub-divided into two research goals. The first goal concerns the solutions needed to make passive testing achievable for industrial practitioners and the second goal concerns the usefulness of the identified solutions.

RG 1. *Propose methods and tools for the industrial application of passive testing at the system level of vehicular systems.*

Research goal 1 (RG 1) concerns challenges such as specification of passive test logic, the required tooling and ways of working with passive testing that can be aligned with industrial software testing processes. This goal directly contributes to the overall research goal.

RG 2. *Evaluate the proposed methods and tools for passive testing concerning practical applicability in an industrial setting.*

Research goal 2 (RG 2) concerns the extent to which the thesis results can be used on industrial data without contradicting current results and what new information the proposed methods and tools provide. This research goal contributes to the overall goal by ensuring that practitioners can use the proposed solution for their data and industrial context.

2 Scope and Limitations

The scope of this thesis is limited in three dimensions: The *topic*, the targeted *system type* and the *software development process scope*. The topic scope is the challenges and solutions encountered when introducing passive testing into an industrial context. Although a research framework for passive testing is presented as one such solution, formal methods, language theory, and mathematical proofs are out of scope for this thesis. The main concern is to present methods and tools that the average tester can use in a real industrial setting. The second dimension is the targeted *system type*. Although the presented ideas may be applied to other system types, this thesis's research has been conducted on vehicular embedded systems. Compared to the ordinary embedded software or even desktop software, an essential property of the studied systems is that the systems are signal-based. Another property is that they are large (many thousands of signals) and consist of many subsystems developed separately and integrated into a complete distributed system. Many of these systems are safety-critical and require testing under more than one condition and system configuration. The third dimension is the software development process scope. Although the proposed methods and tools can probably be used when specifying requirements at any level, the work focuses on testing requirements on the system level.

3 Empirical Research in an Industrial Context

Both regular software development activities and empirical research in an industrial context aim to find industrially viable solutions to industrial problems. However, Basili [4] distinguishes development work from empirical research by the use of structured methods and body of knowledge provided by the academic community. Further, research follows an inductive or analytical paradigm, including a purposeful data collection, solution proposal, and an evaluation of the result. Gorschek et al. illustrate the bigger picture as a journey from problem formulation to deployed technology in their *technology transfer model* [20]. During each step of the model, the researcher selects and applies applicable scientific methods in cooperation with industrial experts. In theory, working with and within the boundaries of this model is straightforward. However, within the work of this thesis, many challenges were encountered. The remainder of this section presents challenges, the technology transfer model, and finally, the journey of this thesis.

3.1 Challenges in Empirical Research

The work behind this thesis has met (at least) two kinds of challenges. The first is the type of solution to the investigated problem. The thesis proposes three types of solutions: An algorithm, a tool, and additions to a software process. The solution type influences the overall challenges throughout the technology transfer model and has had the biggest impact on the level of support required from the industrial partner, particularly for evaluation.

The second kind of challenge concerns the differences between academia and industry concerning Goal, Knowledge and Environment, Size and Complexity, IPR, and Data. The following is a brief description of these challenges, starting with the *goal* of industrial development versus the goal of industrial research. Industrial development aims at developing a product, something to sell. The success criteria concerns aspects like production cost, time to market and price. If other companies can use the solution, it is often avoided by patents and non-disclosure agreements. The novelty of solutions is only meaningful if it results in novel benefits from the customer's perspective or cost-saving. On the other hand, academic research aims to develop knowledge (or artifacts that can produce such knowledge). Here novelty of the solution is an imperative success criterion itself. Other fundamental criteria for an academic are how well the evaluation was performed, how generalizable the results are, and how prestigious the dissemination conference or journal was. The next challenge is *knowledge and environment*. Even when using the best scientific methods, not much can be accomplished without a substantial amount of domain knowledge. In reality, time and resources that can be set aside for a Ph.D. student are limited. Consequently, a ph.D. student spends a great deal of effort in studying documents, tool-chains and development environments. While the Ph.D. student needs domain knowledge to understand the problem and be more independent, the industrial partner also needs knowledge in the chosen research method to set realistic expectations on the outcome. The challenge of *size and complexity* concerns, except for the obvious that industrial systems may be large and complex, the size of the published solutions. The problem, its solution, and evaluation of the solution gets published in an academic conference or a journal, often with a page limit. The goal of being published leads to the next challenge: the view on intellectual property rights *IPR*. Anything published must be anonymized and generalized so it cannot be traced back to the industrial partner or leak any business-critical IPR. Thus, academia has a vast body of knowledge, providing solutions and proof of concepts to *generalized* (small) problems. The last challenge is access to industrial *Data*. This thesis met this challenge by creating a private clone of the entire development tool-chain and a complete simulation environment. Such a sandbox allowed testing the solutions in an actual industrial environment without disturbing the production. The drawback was the significant investment of domain knowledge required to operate it properly.

3.2 The Technology Transfer Model

Figure 3.1 illustrates a journey from problem formulation to deployed technology according to Gorschek et al. [20], the technology transfer model.

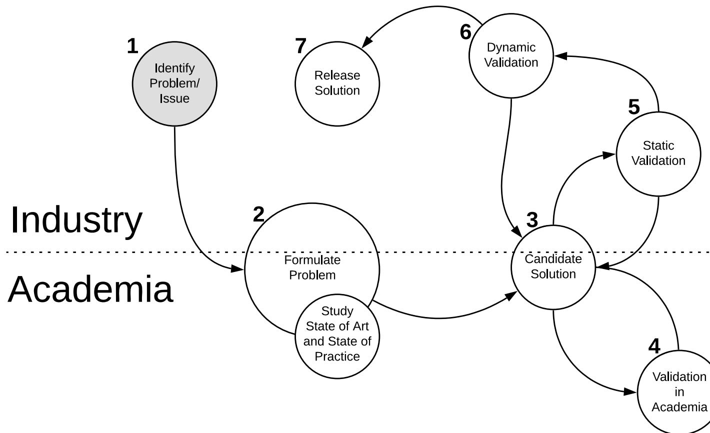


Figure 3.1: Technology Transfer Model [20]

The first step, (1) in Figure 3.1, is to *identify the problem* at hand. In this step, a case study [32, 35] with interviews can be used, for example, in combination with content analysis to reveal issues or problems. The second step, *problem formulation*, concerns formulating the problem to guide the search for related work in the academic discourse. One typical result is a set of research questions. The questions help the researcher to focus when searching the literature for related work. The result of this activity may be a State of the Art (SOTA) that summarizes the academic discourse or a State of Practice that summarizes the industrial solutions. Examples of such methods are snowballing [37], mapping study [29] or a Literature Review [24]. The third step, (3) in Figure 3.1, is the *candidate solution*. Here, the researcher uses the results from step 2 and formulates a first candidate solution. The fourth step, (4) in Figure 3.1, is *validation in academia*. To do this, the researcher may use a limited but representative data set from the company on the solution candidates. Examples of applicable research methods are exploratory case study [32] and a controlled experiment [35]. The fifth step, (5) in tFigure 3.1, is *Static Validation*, where the industry experts examine the candidate solution but not necessarily testing it in practice. This may be done in different ways such as informally, during a number of iterations, as in step 3 or more formally in a case study with interviews. In the sixth step, (6) in the figure, the practitioners test the solution on actual data. Finally, in the seventh step, (7) in the figure, the solution is released internally in the company.

The journey of this thesis through academia and industrial needs

This section goes into more detail about the research process that led to the contributions of this thesis. The focus is on the rationale behind the papers, the

methodological choices, and how the results of the papers have guided the advancement of the process. Further details on what and how the research methods were used are presented in the included papers.

It all started with a concrete problem at Volvo Construction Equipment (VCE) in Eskilstuna. They suspected that testers performed similar test-effort at different levels of integration. A preliminary case study [15] revealed that the problem also occurred at Alstom Transport AB in Västerås in the form of similar test steps repeated in several test cases. This knowledge guided the research of Paper A that capitalizes on knowledge of such similarities to prioritize test case automation. As a mean of reusing test cases between levels of integration, passive testing seemed to be an attractive approach. Based on some further preliminary work in [17] Paper C investigates how testers appreciate the approach and an initial version of the language and the tool at Scania. Since the purpose of the study was to catch the testers' opinions, a case study with interviews analyzed using content analysis was chosen. Paper C's interviews belong to the static validation activity in the technology transfer model, while the tester's trial period (although small) belongs to the dynamic validation. Despite the simplistic language, the testers still found challenges fed back to the candidate solution activity in the technology transfer model. After continuous informal static validation of the updated solution candidates, Paper B adds to the candidate solution(s) and makes a new validation. Dynamic validation of a development tool such as the proposed is challenging to achieve as a Ph.D. student since it requires company resources over quite some time. In this work, a middle way was chosen, where the researcher performed the dynamic validation in the industrial environment and dialog with the practitioners. The work performed in Paper E corresponds to an iteration between the candidate solution activity and the academic validation, gradually refining the proposed solutions concerning false positives. It is completed with a dynamic evaluation of the approach by an expert tester. Finally, Paper D collects the overall results concerning the language.

Chapter 4

Research Contributions

In practice, the research goal of this thesis concerns how to produce passive test cases from software requirements in a way that could be accepted and adopted in an industrial setting. This process is illustrated as a think black arrow at the top of Figure 4.1 between the requirements and the passive test cases. The text above the arrow outlines significant steps in the proposed process to accomplish the translation. The shadowed rectangles in the figure scope individual contributions from the included papers A through F to the research goals (RG1 and RG2). The first goal (RG1) concerns the methods and tools, and the second (RG2) concerns industrial applicability. Finally, the relations between the included papers (yellow document icons in the figure, marked A through F) and the highlighted contributions (C1 through C5) are shown as arrows from the papers to where the contribution is applicable in the process. The remainder of this section outlines these contributions and how they address the research goals. The outline is followed by a series of sub-sections covering more details of the individual contributions.

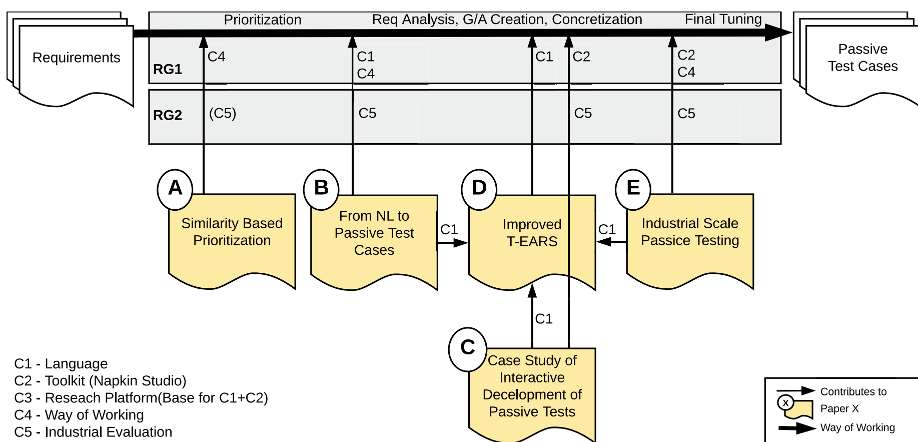


Figure 4.1: Contributions (C1 to C5) and Dependencies of the Included Papers. C3 constitutes the open source implementation behind C1 and C2. The names of the papers are abbreviated for clarity.

The addressed challenges of the first research goal (RG1) include specification of passive test cases, tool support and ways of working. With the work of Independent Guarded Assertions [21] as the starting point, the first challenge is to find an intuitive and effective way to specify the guard that decides when to test and a way to specify the assertion that constitutes the actual test logic. The solution proposed in this thesis is a language and its semantics, T-EARS (Contribution C1). The contributions to T-EARS were collected over time from work with Papers B, C, E and finalized in Paper D. Highlights of the language are further presented in Section 1.1.

Even though the language is designed to be simple and intuitive, practitioners find logical expressions with temporal constraints difficult to interpret and error-prone [16]. Therefore, this thesis proposes an interactive tool for passive test cases (Contribution C2). With an initial prototype in [16] as a starting point, expectations from the testers (Paper C) have been gradually realized in the tool. The tool was also updated continuously with requirements from work with Papers B and E. The tool contribution (C2) is further described in Section 1.2.

The modular architecture of the tool and the language contributions forms a research platform (Contribution C3) for experimenting with the proposed grammar, semantics, and tool features. Scania open-sources the original tool and language prototype[1]. The proposed work in this thesis is a fork[2] of that repository. The work with Paper B through E continuously contributed to the platform as the tool and the language serve as an example of what can be done with the platform. An overview of the research platform is presented in Section 1.3.

The final challenge is to find a way to work with passive testing aligned with existing software testing processes. This thesis presents a way of working (Contribution C4) based on the combined results from Papers A, B, and E. The contribution allows introducing passive testing to complement an arbitrary software testing process to adopt the approach gradually. Paper A contributes to this way of working with an algorithm that helps to order the work of automating test cases. This algorithm has also been applied to order the work of translating requirements to passive test cases based on ordering test scenarios covering those requirements. Paper B contributes with a translation process from natural language requirements to passive test cases. This process is fine-tuned in Paper E, where false fails are addressed by adding further tuning steps to the process. The contributions to the workflow are further presented in Section 1.4.

The second research goal (RG2) is to evaluate the industrial usefulness of the proposed methods and tools. The individual contributions are, in general, evaluated in the paper where they are presented. Paper A evaluates the prioritization algorithm of test cases for automation against other prioritization approaches. Paper B provides a proof of concept by applying the proposed translation process on an industrial requirement. Paper C evaluates the approach, language and the tool from a testers perspective. Paper D evaluates the expressfulness of the final language by analyzing a large set of safety critical requirements. Paper E uses and evaluates the whole resulting way-of-working process (except requirement prioritization) for a set of safety-

[1] https://github.com/scania/saga

[2] https://bitbucket.org/danielFlemstrom/napkin

critical requirements. This and relevant results from Papers A-D are summarized in Section 2.

1 RG1 - Methods and Tools For Passive Testing

This section outlines the contributions that together fulfill the first research goal (RG1), to "*Propose methods and tools for the industrial application of passive testing at the system level of vehicular systems*". Section 1.1 outlines an intuitive language including templates for intuitive specification of passive test cases. The resulting development environment is outlined in Section 1.2 and Section 1.3 describes the open-source research platform behind that tool. Finally, Section 1.4 presents a way of working with passive test cases in an industrial context.

1.1 C1: An Intuitive and Easy To Use Specification Language

T-EARS was first introduced in [17] as a specification syntax for independent guarded assertions (G/As) [21, 31]. The G/A approach in [21] separates the passive test case into a guard, deciding when to test and an assertion expression, deciding what to test. The improved T-EARS proposed in this thesis is the collective result from the work of all the included papers. While the details are presented in Paper D, this section gives an overview and highlights selected features of the language.

```
1 'Bp-1' = while true                 shall <system response state A>
2 'Bp-2' = while <system state G > shall <system response state A> within t
3 'Bp-3' = when  <events G> shall <system response state A> within t
4 'Bp-4' = when  <events G> shall <response events A> within t
5 'Bp-5' = when  <events G> shall <system response state A> for tf within tw
6 'Bp-6' = when  <events G> shall <system response state A> within tw for tf
```

Listing 4.1: Resulting Boilerplates

Boilerplates: The boilerplates in Listing 4.1 ensure the tester a good starting point when writing a test case. Their purpose is also to rule out confusing or ambiguous expressions. The boilerplates, or templates, are closely related to the patterns identified in Easy Approach to Requirements Specification (EARS) [26]. In fact, Table 4.1 shows a more detailed mapping, how to use the boilerplates to realize the EARS patterns. The `while` keyword suggests that the guard is described by a system *state* expression and the keyword `when` that the guard is described by a system *event* expression. The expected system response is described by the `shall` keyword followed by an *assertion* expression. The assertion expression may describe either an *event* or a *state* response depending on the boilerplate. The keywords `within` and `for` specifies timing constraints and allowance, further described in the timing section below.

Types: An important contribution of this thesis is the more strict use of types in T-EARS. In the boilerplate section, system states and events were discussed. States are represented by the Intervals data type in T-EARS and can be specified manually, as shown in row 2 in Listing 4.2, or by an expression as shown in row 3. Essentially the Intervals data type consists of a series of $[t_{start}, t_{end}]$ for each interval the system

is in a particular state. System events are represented by the Events data type, a series of points in time where a particular event occurred.

Besides Intervals, and Events there are Signals, which are essentially a series of [timestamp, value] pairs. Signals are used for creating Intervals or Events together with trivial types such as Boolean, Float, and Time.

```
// Intervals
def intervals door_is_open = [[12s.30s],[90s,120s]] // manually defined, or
def intervals door_is_open = door_position > 2     // using signal expression

// Events
def events door_opens = [12s,90s]      //manually, or
def events door_opens =
    rising_edge(door_is_open == true) // Events expression
def events door_closes =
    falling_edge(door_is_open == true)
```

Listing 4.2: Examples of T-EARS Types and the def Structural Element

Expressions (logical): The Signal, Interval, and Event data types can be combined into logical expressions such as two system states that should be fulfilled at the same time. A practical example is the expression "engine on" AND "gear in reverse". Expressions can also be used for specifying a system state from signal expressions, e.g. $S_1 > 1$ *and* $S_2 > 5$ would yield a series of intervals where the signal value for S_1 is above 1 at the same time as the signal value for S_2 is above 5.

Expressions (time): In T-EARS, the temporal specification has been restricted to a minimum to keep the resulting expressions intuitive. As a result, the case

EARS Pattern		T-EARS Realization
Ubiquitous: *A* shall always hold	BP1	`while` true `shall` *A*
State pattern: `while` preconditions `the` system name `shall` system response	BP2	`while` G `shall` A `within` t
Event Pattern: `when` preconditions `the` system name `shall` system response	BP3 BP4	`when` G `shall` A `within` t
	BP5	`when` G `shall` A `for` t_1 `within` t_2
	BP6	`when` G `shall` A `within` t_1 `for` t_2
Option: `where` expression [requirement]		`where` B [Any BP1-6]
Unwanted Behavior: `if` unwanted behavior `then` `shall` system name system response		[Modeled using any BP1-6]
Complex: Mix of above patterns		Accomplished by logical conjunctions of G or A

Table 4.1: How the Resulting T-EARS Boiler plates Realizes the EARS Patterns. *G* denotes a Guard Expression and *A* is the Assertion Expression

of filtering intervals of a particular length has been separated from creating an event whenever an interval is long enough. As an example, the keyword `for` always produces an event each time an interval has exceeded a specified amount of time. Another example of time expressions in T-EARS is the `within` keyword that specifies an acceptable system response delay in relation to the guard.

Expressions (filter): As previously discussed, the use of the `for` keyword would have different semantic meanings depending on where it occurs in an expression. Therefor the keywords `longer than, shorter than` are used for filtering intervals longer or shorter than a threshold.

Structural Elements: Structural elements are used for increasing the readability of the expressions. Constants can be defined to replace numbers with a meaningful name. The structural elements also have an integral part of abstract vs concrete signal handling. Using the `def` as shown in Listing 4.2, abstract signals can be mapped to example signals, or concrete expressions of a particular system release. In the listing, the `def` keyword followed by either `intervals, events` binds the expression to the right to the identifier and ensures type-safe operations. Paper D also gives examples on how the structural elements can be used for increasing the density of the log files by logging whole telegrams, carrying several binary signals. The resulting state from the signals can then be extracted as Intervals using T-EARS expressions.

1.2 C2: An Interactive Passive Test Case Development Environment

This section gives an overview of the main functional areas of the proposed interactive development environment for passive test cases: Napkin Studio[3] . The section also provides a brief discussion on how the different functions can be used together when writing passive test cases. Napkin Studio is a fork of the SAGA Tool[4]

Napkin Studio is web based and can be used as a stand-alone tool or be integrated with a test result database view by forming an URL specifying the passive test case, log file, and the definition file to load. When used stand alone, a file selection dialog allows browsing the local file system for a log file in the `jsondiff` format. The tool loads the log file and lists all logged signals in the Signal list (nr (1) in Figure 4.2). Since there may be many signals in a log file, a filter (nr (2) in Figure 4.2) allows the tester to reduce the list only to show the signals of interest. The tester can then drag a signal from the Signal list (nr (1) in Figure 4.2) and drop it on the Plot Area (nr (6) in Figure 4.2) to examine the signal. Another way to examine or edit the singal is to drop it on the Signal editor (nr (3) in Figure 4.2). It is also possible to create new signals in the signal editor, using an existing signal as a template or one of the provided signal generators. Several signals can be viewed and edited at the same time by adding mini editors to the right pane (the "Add mini editor" button in Figure 4.2). The Snippet Editor section (nr (4) in Figure 4.2) allows the tester to develop the expressions interactively using the loaded or created signals. There

[3]https://bitbucket.org/danielFlemstrom/napkin (contact daniel.flemstrom@mdh.se for access)
[4]https://github.com/scania/saga

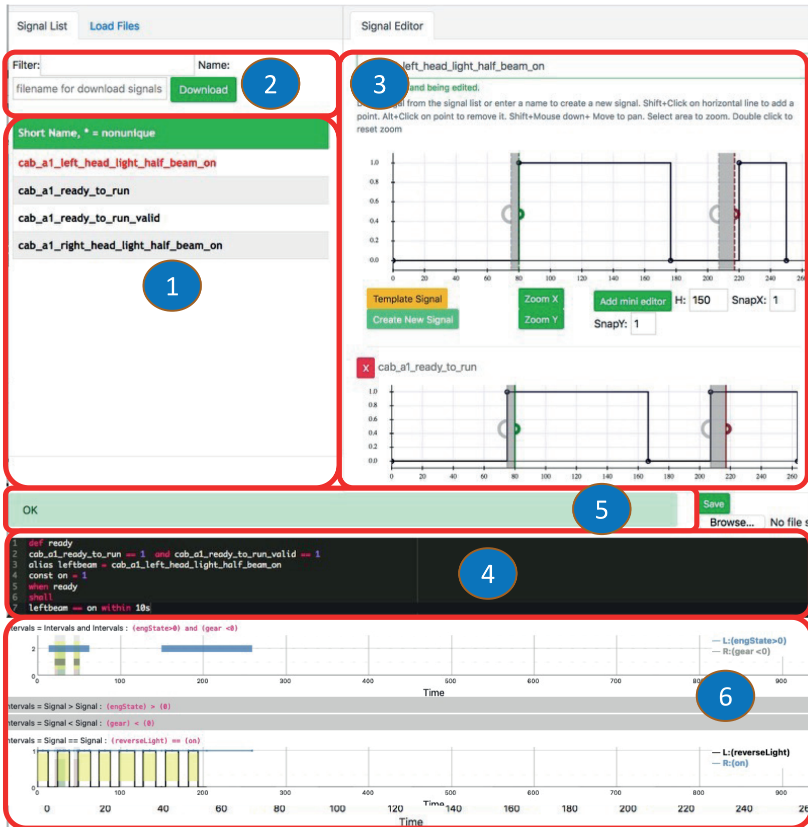


Figure 4.2: Napkin Studio Functional Area Overview

is also a larger version of the editor in the tabs of the left pane. Each time the user presses ctrl+enter, the current T-EARS expression in the editor is evaluated. Feedback on syntax, interpretation, and missing signals is provided by a status bar (nr (5) in Figure 4.2), and the evaluation result is overlaid in the Signal editor (nr (3) in Figure 4.2) and the Plot area (nr (6) in Figure 4.2). The Plot area also visualizes each sub-expression.

Examining and Comparing Execution Log Files

In a signal based system, all communication between program units as well as sub systems is done via signals that can, to a varying extent, be logged during execution. These signals, e.g, digital or analog I/O signals captures the system input, the internal state of the system and the output over time. The proposed approach to passive testing relies on observing these signals, both to determine when to test and for the actual test logic. Examining such logs is thus an important part of understanding the system behavior. Napkin Studio allows such log files to be loaded

using different formats and sources. If the user wants to compare two or more signal files, a synchronization strategy must be provided. Currently there is a default implementation available treating the first sample as time zero.

The Signal List (marked as nr (1) in Figure 4.2) provides information such as a short name, size of the signal, and the full name (the original name of the signal together with the log file name to uniquely distinguish each signal). If a signal name appears in more than one loaded log file, the user can compare the signals. Any differences between the signals are shown as fail intervals and the equal parts as pass intervals using a generated A T-EARS expression that shows up in the snippet editor. The user may also select signals from the signal list to show in the Plot Area (marked as nr (5) in Figure 4.2). Aside from the Plot Area's standard panning and zooming features, signals can be skewed in time by adding a positive or negative latency. Sometimes the tester needs to find out the cause of a signal change. This is possible by letting the tool automatically find and view all other signals that have changed during the visible interval of the currently viewed signal. There is also a signal generator that can be used for analyzing signals as shown in Figure 4.3. The "Learn From Current Signal" button (nr (1) in Figure 4.3) analyses the currently edited signal and lists the different values together with their sample probability (nr (2) in Figure 4.3). The leftmost fields (nr (3) in Figure 4.3) shows and controls the timing of the generated samples. The emission cyclicity shows the mean value of how often the signal has changed. A high jitter means that the cyclicity is not evenly

Figure 4.3: The Signal Generator. The generator can be used for generating new signals or analyzing existing signals.

distributed across the log file.

Finally, using the snippet window (marked as nr (4) in Figure 4.2), T-EARS can be used as a log query language to mark out interesting regions of a log file. The T-EARS expression `while voltage>10` would mark out all intervals where the value of the signal *voltage* is above 10. Each such hit would also be listed in the context menu. Selecting one such region zooms the Plot Area and centers the timeline around the selected region.

Working With Implementation Analysis and Signal Interfaces

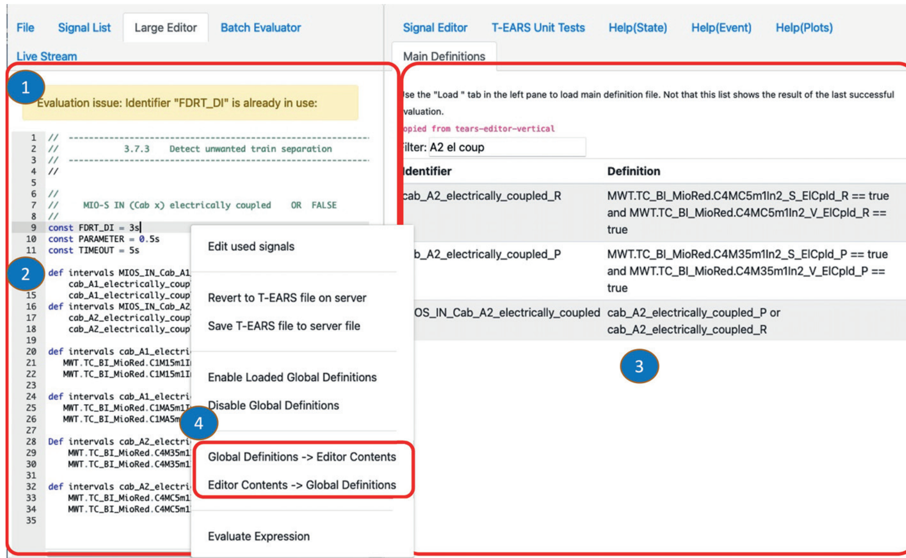


Figure 4.4: Editing a Main Definitions File

Signal names in industrial systems are often long and hard to read since these names often contain compressed information on the physical location of the signal source. Examples of such information are busses, sensors, or program modules. During development, such information typically changes due to reallocations of signal routes or functional updates. In addition, the requirements often define signals on a higher level of abstraction. Where the requirements mention the door open signal, this may actually be realized by an expression involving many signals in the current implementation. Napkin Studio allows hiding all such details in the “main definitions” block or file. The main definitions block consist of a series of T-EARS expressions that define constants, aliases, or more complicated expressions the tester wants to hide. Typically these definitions are common to many passive test cases. When active, the main definitions are evaluated before the current T-EARS expression. These definitions are created using the regular (large version) of the expression editor as shown in Figure 4.4. The expressions (nr (2) in Figure 4.4.) end up in the

main definitions search view to the right (nr (3) in Figure 4.4). The test engineer also uses the latter to find already defined abstract signals to use when translating requirements to passive test cases. Using the context menu (nr (4) in Figure 4.4), the definitions can be moved between the evaluation context and the editor.

Working With Requirements or Passive Test Cases

Napkin Studio allows a uniform way of working with requirements, abstract passive test cases, and concrete passive test cases. The difference between an abstract test case and a concrete test case is how the signals are defined. Preferably, only the abstract signals[5] defined in the requirements are used in the expressions. For a requirement (or abstract test case), the user can create hand-made example signals of situations where the test case should pass or fail. Such hand-made signals are valuable for validating the expressions' correctness since the current T-EARS expression is evaluated with each change of the created signal(s). Such examples may also help a test engineer to understand the intention behind a requirement. This is illustrated in Figure 4.7. There are also a few signal generators, as shown in Figure 4.6 and Figure 4.3, that can generate random signals to verify that the expressions work for unexpected corner cases. These signals can be saved together with the test case to distribute or archive the examples.

An abstract passive test case becomes concrete by mapping the abstract signal names to T-EARS expression. This mapping is typically done in a separate file containing the "main definitions" block as illustrated in Figure 4.4. In fact, when the test engineer translates a requirement to a passive test case, one of the most essential tools for the test engineers is the main definitions search view (nr (3) in Figure 4.4). This search view allows reusing already defined abstract signals. This information is also accessible via the editor's auto-completion, shown in Figure 4.5. The auto completion function has been adapted according to the needs of the test engineers in two distinct ways. First, a fuzzy search is done to match identifiers. The string `stinh` (the first letters in each word of ***st**art **inh**ibit*) would match as shown in Figure 4.5. Second, the auto-completion information not only shows the matching identifiers but how that identifier is defined (e.g., a signal, alias or constant). This is important since there may be abstract signals with similar names that only differ in some specific implementations.

When developing larger passive test cases the larger Editor in the left pane of the window can be used for the test case and the snippet-editor in the middle of

[5] Spaces and special chars may be removed to prevent misspells when used in logical expressions.

```
while stinh
    start_inhibit_bypassed_A2 = (Intervals) MWT.TC_BI_MC50.C4MC5m1In11_S_StaInhOvr == true and    MWT.TC_BI_MC50.C4M
    start_inhibit_bypassed_A1 = (Intervals) MWT.TC_BI_MA50.C1MA5m1In11_S_StaInhOvr == true and    MWT.TC_BI_MA50.C1
    noStaInh = (Timeout) 0                                                                        const global
    MIOS_IN_Start_inhibit_is_bypassed_in_A2 = (alias) start_inhibit_bypassed_A2                   alias global
    MIOS_IN_Start_inhibit_is_bypassed_in_A1 = (alias) start_inhibit_bypassed_A1                   alias global
    CCUO_IP_Start_inhibit_reasons = (alias) MWT_traction_block_reason                             alias global
    GWS_WTB_IN_Start_inhibit_from_remote_consist = (alias) MWT.TC_BI_iCCUS_AToCCUO.DBC_MV_SX_StaInhRsRmCst alias g…
    GWS_WTB_OUT_Start_inhibit_to_remote_consist = (alias) MWT.TC_BI_iCCUS_AToCCUO.DBC_MV_SX_StaInhRs   alias global
```

Figure 4.5: Editor Auto Completion

Generate Pulse Train
Start Time (s): 0
Average Pulse Length (s): 10
(Approx.) End Time (s): 167.6577954196082
Pulse Length Jitter (s): 5
Emission cyclicity (s): 15.241617765418926
Low value: 0
Emission cyclicity jitter (s): 14.161403357991611
High Value: 1
Approx. nr of samples: 11
Learn From Current Signal
Force Overwrite Generate

Figure 4.6: Signal Pulse Train Generator

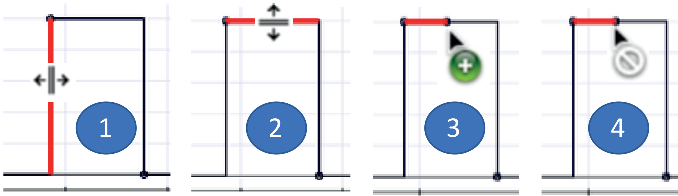


Figure 4.7: The Four Edit Modes of the Signal Editor: 1) Time-shift Sample, 2) Change Level, 3) Add Sample, 4) Remove Sample

the screen (nr (4) in Figure 4.2) can be used to develop individual expressions. To support the work of understanding or debugging more lengthy expressions, each sub expression is automatically added as a plot in the Plot Area at the lower part of the screen (nr (6) in Figure 4.2). Each plot shows the used grammar rule and types matching the expression along with any actual arguments. The result of the T-EARS expression in the currently active editor case is overlaid each plot. Further, the user can add other potentially interesting signals to study together with the sub expressions.

Tuning Passive Test Cases

In a traditional test case, the system's state is well known during each point of the test sequence since the test script specifies all actions taken to enter that state along with the actual test logic. In the case of passive testing, there is no trivial way of knowing the system's current state or what previous actions brought us here. The state of the system needs to be determined by observing the right set of signals. As a consequence, if the expression specifying when to test is not complete, a passive test case may evaluate as expected on one log file but would fail on another, even if the system behaved correctly in the first log. One reason may be undocumented dependencies in the requirement. A concrete example is a specification of a brake

light, where the requirements engineer forgot to write that, aside from the brake pedal, there are other systems that may order a brake light. A test case ensuring that the brake light is not lit when the brake pedal is un-touched would then fail mysteriously for some execution logs. There are also other sources of false fails. During the work with Paper E, the tester's concern about the risk of false positives [16] was confirmed. In response, an overview showing the working set of passive test cases evaluated over a set of representative log files was introduced to the tool set (Figure 4.8a and Figure 4.8b). In the paper, the log files came from known passed test cases. In reality, each fail needs to be closely examined to ensure that the fail is not due to a real fault, which brings us to the next section, using the tool for examining the result and potential faults.

Analyzing the Results

There are currently three options for executing passive test cases. The first has already been discussed and is the evaluation of the T-EARS expression directly in the Editor. The second is found in the "Batch Evaluator" tab of the Napkin Studio and allows evaluating a set of passive test cases over one log file. This view was used in the paper E to show the requirement coverage of an expert test execution as shown in Figure 4.8a. The view shows whether each passive test case (G/A) passed (P), Failed (F), or could not be run due to some reason (-). As seen in the Evaluation Details column, the number of activations, passes, and fails are also shown. The column also shows details on missing signal or syntactical problems in the G/A.

G/A	Result	Evaluation Details
REQ-244	P	2 Guard Activations,2 passes , and, 0 fails
REQ-245	F	2 Guard Activations,3 passes , and, 1 fails
REQ-246	F	1 Guard Activations,2 passes , and, 1 fails
REQ-248	-	Guard never activated
REQ-253	-	"...._NoTcmsEmBr" is not logged.
REQ-254	-	"...._NoTcmsEmBr" is not logged.
REQ-255	-	"...._NoTcmsEmBr" is not logged.
REQ-258	-	"...._NoTcmsEmBr" is not logged.
REQ-259	-	"...._NoTcmsEmBr" is not logged.
REQ-260	-	"...._NoTcmsEmBr" is not logged.
REQ-281	F	2 Guard Activations,2 passes , and, 1 fails
REQ-283	-	"...._NoTcmsEmBr" is not logged.
REQ-290-A1	-	Guard never activated
REQ-290-A2	-	Guard never activated
REQ-349	-	"...._NoTcmsEmBr" is not logged.
REQ-350	-	"...._NoTcmsEmBr" is not logged.
REQ-456-A1	F	1 Guard Activations,2 passes , and, 1 fails
REQ-456-A2	-	Guard never activated

(a) Many G/A – one Log File

Regr Log G/A	TC-001	TC-003	TC-005	TC-007	TC-011	TC-013	TC-016	TC-017	TC-018	TC-019	TC-020	TC-021	TC-024	TC-068
REQ-244	P	P	F	P	P	P	[P]	P	P	P	P	F	[P]	P
REQ-245	-	-	-	F	-	-	[F]	-	-	-	-	-	-	-
REQ-246	-	-	-	-	-	-	-	[P]	-	-	-	-	P	P
REQ-248	-	-	-	-	-	-	-	-	[F]	-	-	-	-	-
REQ-253	-	[F]	-	-	-	-	-	-	-	-	-	-	-	-
REQ-254	-	-	[P]	-	-	-	-	-	-	-	-	-	-	-
REQ-255	-	-	-	[P]	-	-	-	-	-	-	-	-	-	-
REQ-258	[P]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-259	[F]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-260	-	-	-	-	[F]	-	-	-	-	-	-	-	-	-
REQ-281	P	P	F	P	F	P	-	-	-	-	-	[F]	F	-
REQ-283	-	-	-	-	-	-	-	-	-	[F]	-	-	-	-
REQ-290-A1	-	-	-	-	-	-	-	-	-	-	[F]	-	-	-
REQ-290-A2	-	-	-	-	-	-	-	-	-	-	[F]	-	-	-
REQ-349	F	F	F	F	F	[F]	F	F	F	F	F	F	F	F
REQ-350	-	-	-	[F]	-	-	-	-	-	-	-	-	-	-
REQ-456-A1	-	-	-	-	-	-	-	F	-	-	-	-	P	[F]
REQ-456-A2	-	-	-	-	-	-	-	P	-	-	-	-	-	[F]

(b) Many G/A – Many Log Files

Figure 4.8: Evaluation Summary Views [Paper E]

The third option is the "many G/As - many logs" overview created by the back-end server. A working set of passive test cases, a main definition file, and a set of log files are collected to an overview with clickable links as shown in Figure 4.8b from Paper E. This view gives a good overview of the requirements coverage of a regression suite or a higher level smoke test. In paper E, the log files were taken from regression tests specifically designed to test particular requirements. The corresponding G/As

are marked with brackets in the Figure. Notably, there is not much activation of the G/As outside the expected ones, however, passive testing allows creating new logs with actions that potentially activates many more requirements at the same time, which would fill up the matrix better.

If a G/A fails for a log file, the user clicks on corresponding cell and is directed to an HTML page where Napkin Studio, the corresponding log file, G/A, and main definition file are automatically loaded. Primarily, the tester would now use the Plot Area (nr (6) in Figure 4.2, with a close up in Figure 4.9) where the evaluation of the selected G/A over the selected log file is shown.

Figure 4.9: Plot Area With Activated Plot Context Menu

Right-clicking on the evaluation brings up the context menu (shown to the left in Figure 4.9). The context menu lists the failed events or fail regions of the current passive test case either as a list (nr (1) in Figure 4.9) or as a hierarchical structure (nr (2) in the figure). Selecting e.g the bottom fail region (nr (1) in the figure) will center and zoom in to the failed region as shown at nr (3) in the figure. After further zooming or panning, the user can choose to add all signals that have changed during the visible time period, to the plot area. This is done using the context menu (the choice at nr (4) in the figure). The signals are sorted concerning number of changes and added to the Plot Area for further analysis. Signals in the Plot Area can also be shifted by adding a latency, or edited in the Signal Editor to find out what caused the failure.

Finally, removing the `within` statements from a G/A can also give important information on how the system behaves concerning latencies and delays.

Alstom Transport AB

1.3 C3: Research Framework

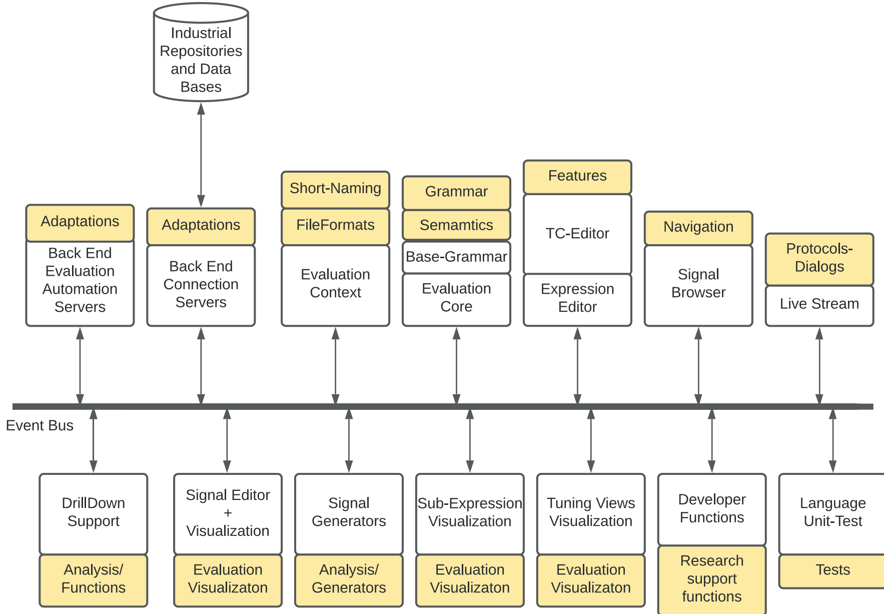


Figure 4.10: Resulting Research Framework

Figure 4.10 illustrates the research framework that constitutes contribution C3 in this thesis. The framework consists of prototype implementations for the basic modules (white shapes in the figure) of an intuitive passive test case specification tool. The language (contribution C1), its evaluation core, and the Napkin interactive studio(contribution C2) are denoted by the yellow parts in Figure 4.10. The framework is designed to be modular and easy to extend, providing a solid base for adding and evaluating grammar constructs, visualization of various connections to industrial information systems. The framework is realized by a web-based editor and three back-end servers. While the back-end servers are implemented in python, the web-based editor is implemented in javascript and Vue2.0, and Bootstrap-Vue. The grammar is expressed with rules in the Ohm grammar language, and a javascript function realizes the semantics of the rule(s). Adding a grammatical construct can quickly be done by extending this OHM grammar and adding the corresponding semantic action to the evaluation core. The *evaluation core* can also be used remotely via the evaluation REST server. RESTFul evaluation allows passive test cases to be evaluated en masse or over large log files at the server-side. There is also a RESTFul live-view server currently communicating with the Alstom Transport AB logging system. The view is updated every 2 seconds, and the expression is re-evaluated over the whole log file, just as for a loaded log file. Finally, since RESTFul servers are less efficient for data-intense applications, a web-sock-server implements saving /

and storing the edited passive test cases and loads log files.

The prototype follows a modular architecture with a message bus for synchronization. For efficiency, the evaluation context that contains the information on the currently loaded signal log is shared between all components. Whenever an expression or signal is changed, a request for evaluation is sent on the message bus, the passive test case is evaluated on the loaded log, and the result is sent out on the message bus. All components subscribing for that message can then take proper action. This architecture makes it easy to extend the default modules that for most features of an Integrated Development Environment(IDE) for passive test cases by adding custom modules that listens to, or sends messages on the, message bus. While there are other open source IDE´s such as Eclipse or Visual Studio Code, NAPKIN Studio covers the signal based aspects of the development.

Figure 4.10 shows the logical structure of the functionality of the prototype implementation. The white boxes denote the default implementation, and the yellow boxes suggest adaptations or areas of experimentation.

Example of Research Platform Utilization

This section demonstrates how the proposed research platform can be used to try out an alternative syntax for specifying test cases. The syntax is a GIVEN-THEN-WHEN format used as an experiment by Alstom to find new optimal ways of specifying requirements for their Train Control and Management System (TCMS). The syntax is inspired by, e.g., Cucumbers' Gherkin [6]. A requirement in this format may look like Listing 4.3.

```
1 GIVEN Train is at standstill [ No zero speed = Zero speed ]
2 WHEN At least one door is not closed [ All doors closed = false ]
3 THEN TCMS shall inhibit traction [ Enable traction = false ]
```

Listing 4.3: Original Requirement Text

After a mild rewrite to a passive test case and some minor reformatting to facilitate parsing, the expression would look like Listing 4.4.

```
1 const Zero_speed = 1
2 GIVEN No_zero_speed == Zero_speed  // Train is at standstill
3 WHEN rising_edge(All_doors_closed == false)  //At least one door not closed
4 THEN  rising_edge(Enable_traction == false)  //TCMS shall inhibit traction
```

Listing 4.4: Prepared for Evaluation

Analyzing the expression reveals that the expression in Listing 4.4, in fact, is a guarded assertion with an extra (interval) guard condition defined by the GIVEN clause. The grammar for a guarded assertion is already implemented in the framework, so the extra condition can be added as an optional rule in the existing OHM main rule for guarded assertions as shown in Listing 4.5. The listing also shows that THEN is added to the existing SHALL rule since they mean the same thing.

```
1 GA = ((identifier "=")? Config? GivenContext? GuardedAssertion)
2 shall =  (caseInsensitive<"shall"> | caseInsensitive<"then">)
```

Listing 4.5: Grammar Additions. An optional GivenContext rule is added to the GA grammar rule. The shall keyword rule is extended to accept either shall or then

[6]https://cucumber.io/docs/gherkin/reference/

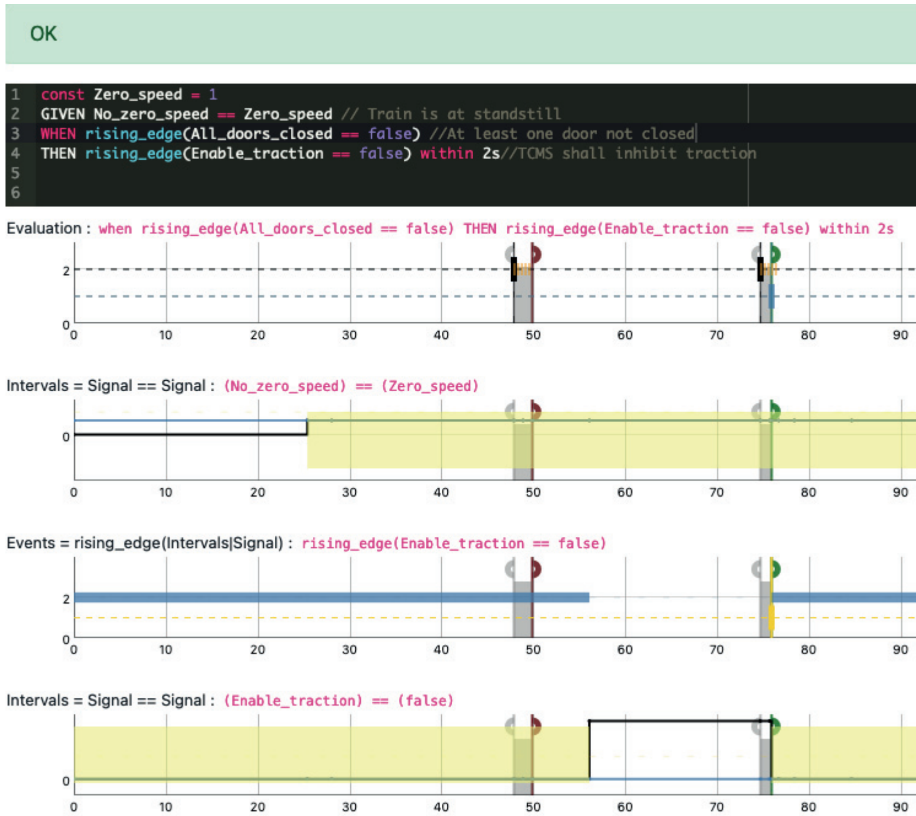


Figure 4.11: Resulting Evaluation of GIVEN-WHEN-THEN Expression With Example Signals in Napkin Studio

The GivenContext OHM-rule requires a corresponding javascript semantic rule. The Napkin framework already contains all necessary rules and implementation to unleash the full power of Intervals expressions. The new function for the GIVEN semantics thus delegates the expression evaluation as shown in Listing 4.6.

```
1 GivenContext:function( _given, interval_guard){
2   return  interval_guard.eval();
3 }
```

Listing 4.6: Semantic rule for GivenContext

The semantic rule of the main GA rule is modified to save the result of the GIVEN expression in the current context (lines 3-4 added to Listing 4.7).

```
1 GA:function( identifier, _eq, config, given_context, guardedAssertion){
2   var name = evalOptional(identifier,"");
3   var given_context = evalOptional(given_context,undefined);
4   moduleContext.currentGiven = given_context;
5   ...
6 }
```

Listing 4.7: Semantic rule for GivenContext

Finally, code is added to the state and the event guard evaluation to make use of the extra GIVEN guard in Listing 4.8. The resulting guard expression is now the optional GIVEN expression AND the guard.

```
GuardedAssertion_eventGA: function(_when, guard, _shall, _verify, assertion)
    {
    ...
    if (moduleContext.currentGiven != undefined){
      var SR = moduleContext.currentGiven.value;
      G = operators.and(GE.value,SR);
      if (typeof moduleContext.currentGiven.plots != 'undefined'){
        plots = moduleContext.currentGiven.plots;
      }
    }
    else{
     G = GE.value;
    }
    . . .
 }
```

Listing 4.8: Code added to the Semantic rule of the (event) guarded assertion

Although the resulting grammar, in theory, allows writing funny looking expressions mixing T-EARS and the new format, it allows quick experimentation with full support of code completion, creating signals, as well as examining sub expressions. As a bonus, the GIVEN-WHEN-THEN can also be used as GIVEN-WHILE-THEN to properly capture system state requirements, without adding anything to the

Figure 4.12: Creating "Unit tests" For the New GIVEN-WHEN-THEN Grammar in Napkin Studio

OHM-grammar nor the javascript semantics code.

When testing out the new grammar, there is a T-EARS Unit test tab on the rightmost pane as shown in Figure 4.12. The expression and its description can be entered while the expression is continuously evaluated while typed in. The expected result may either be OK, an evaluation error (due to undefined signals), or syntax error. The latter is used for checking that forbidden expressions stay that way.

1.4 C4: Way of Working

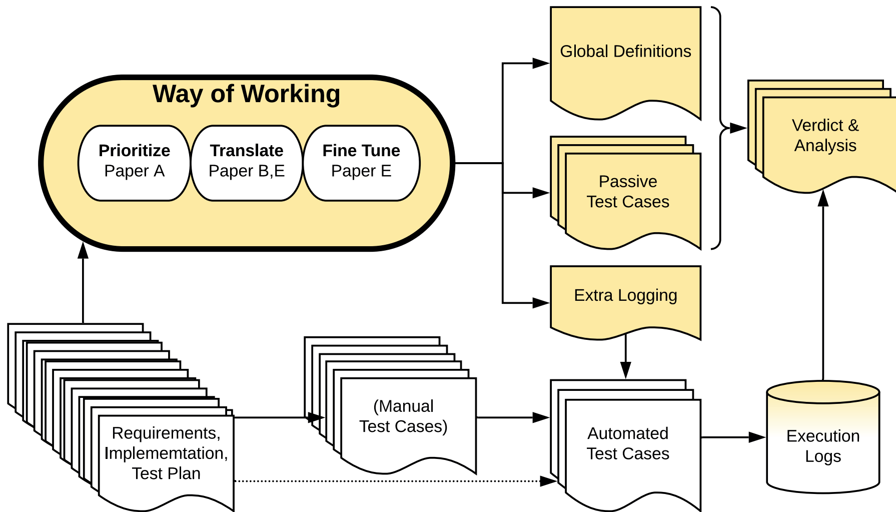


Figure 4.13: Generic Software Testing Workflow. The yellow parts shows suggested additions to the overall software testing workflow and its Artifacts.

Contribution C4 combines Paper A, B, and E results to form a complete way of working when translating each requirement to one or more passive test cases. From Paper A, a prioritization algorithm is adapted to prioritize scenarios for a function. The scenarios are used as a scope of requirements to be translated into passive test cases. Paper B contributes with an overall translation process from a selected function with requirements to a concrete passive test case. Finally, the contribution from Paper E addresses some of the shortcomings identified in Paper C regarding false positives and concludes the translation process with some fine-tuning to reduce false positives.

Figure 4.13 shows how these contributions can add to an existing software testing workflow. The traditional flow of artifacts (bottom row in Figure 4.13) starts with requirements, a test plan, and an implementation (SUT). From this information, manual or automated test cases are created. These test cases can be used for producing execution logs. The proposed way of working (upper row in yellow) starts with the same information but generates, in addition, the yellow artifacts to the right in the figure. These artifacts include one or more passive test cases for each

requirement and any global definitions or extra logging required to evaluate the passive test cases on the execution logs.

Prioritizing the work

Both traditional (active) automated test cases and passive test cases are created over time from the requirements in the scenarios specified by the test plan. During this time, one or more system releases may occur that require re-testing of the system. Thus, intuitively, a large amount of automated test cases as early as possible is beneficial. Further, for safety-critical functions, some requirements should be tested in several scenarios. For example, consider two scenarios of a brake light when the brake pedal is pressed: One scenario concerns a vehicle standing still, and another scenario may concern a vehicle approaching another car. Unfortunately, these scenarios are traditionally translated to manual or automated test cases without considering reuse of already translated test logic. The test for the brake light is repeated in the test cases for both scenarios.

A passive testing approach would allow writing the test logic for each requirement and reusing it for the remaining scenarios. As a bonus, all produced test logic can remain active during other scenarios, contributing to a higher requirement coverage throughout the entire testing process.

The proposed algorithm orders the scenarios to achieve as many parallel passive test cases as possible as early as possible. The assumption is that requirements tested by one scenario are completely translated before translating the following scenario begins. The process is outlined in Figure 4.14. The figure shows how the new

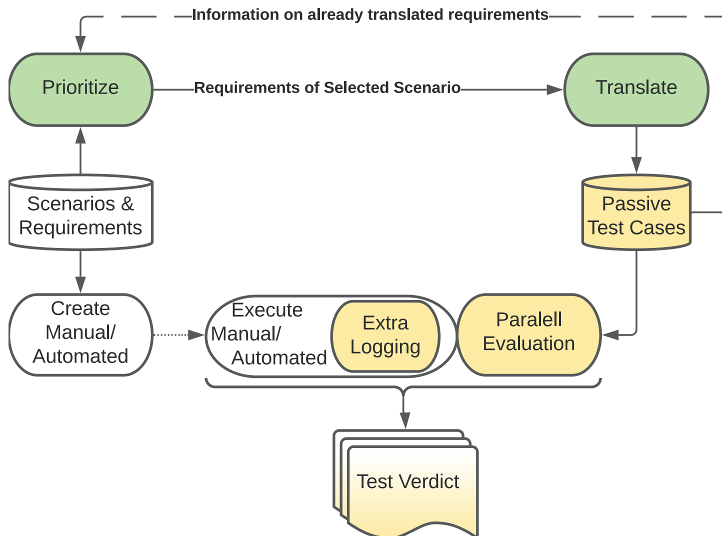


Figure 4.14: Prioritized Translation Activities(green) and Artifacts (yellow) in relation to a Generic Classic Testing Process (black/white)

activities *Prioritize* and *Translate* hooks into a traditional software testing process, producing *passive test cases* and *extra logging*. Whenever any regular (manual or automated) test cases are executed, these passive test cases can be evaluated over the execution logs from those regular test cases. In more detail, the activity *Prioritize* selects one scenario, and the activity *Translate* (further outlined in Section 1.4) creates one or more passive test cases from each requirement in the selected scenario if the requirement is not already translated. Each passive test case also adds to a list of signals that need to be logged, denoted as the extra logging in the figure. This extra logging is applied to all existing manual or automated test cases, allowing a parallel evaluation of all available passive test cases for all test cases.

Before going into details on how the selection is made, consider the example in Figure 4.15

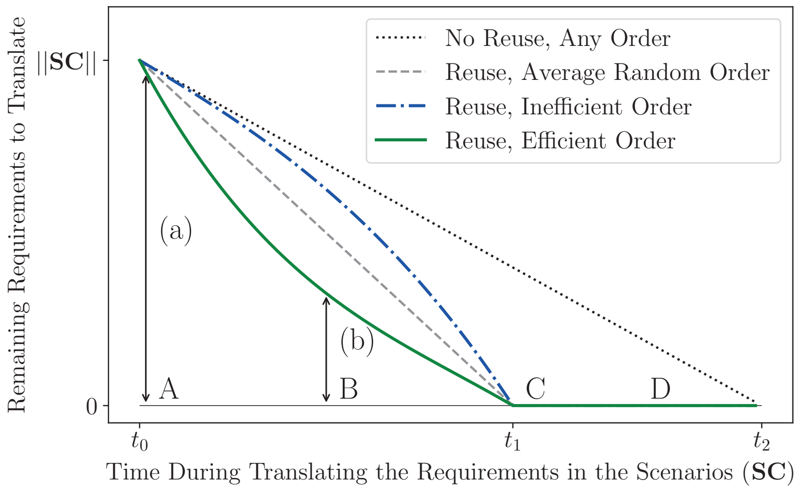


Figure 4.15: Successive Translation of Requirements Using an Adapted Prioritization Algorithm from Paper A with four test sessions at A,B,C, and, D)

Figure 4.15 illustrates a situation where the requirements of a set of scenarios are successively translated to passive test cases during a development project. The work starts at t_0 and is completed at t_1 if requirements are only translated once (in the scenario where they first occur, and implicitly reused for the other scenarios). If such reuse is not considered, the work will continue until t_2. The different slopes show four examples of how the number of remaining requirements to translate is reduced over time given different prioritization strategies. Since this process takes considerable time (t_0 to t_1), a release may be tested before the work is completed. These testing occurrences are illustrated in the figure with the points A through D.

Point A, in the figure, illustrates a first test of the complete system (e.g., an early release with a new component that is integrated). Notably, few requirements have been translated into passive test cases. At point (B), however, things get more interesting. Some of the requirements have been translated, and some not. Notably,

the remaining (un-translated) requirements are fewer than if we had not used the prioritization. The benefit of using prioritization is the difference between the dotted line (No Reuse, Any Order) and the slope (Reuse, Efficient Order). At point C, all requirements have been translated. Finally, point D shows where that there is no more benefits after point D concerning the selected prioritization.

Definition 1. *(Derived From Paper A) The p/e value for a scenario SC is given by:*

$$p/e(SC) = \frac{gain + potential}{effort} = \frac{|SC| + \sum_{s \in S_A} O[r]}{\max(|R_A|, 1/\infty)}$$

$$R_A := s \in \tau : O[s] > 0$$

Where SC is the currently evaluated scenario ,|SC| is the number of requirements of scenario SC, $O[r]$ is the (remaining) requirement occurrence count map for the requirement r, and R_A is the set of requirements that requires translation.

As presented in Paper A, the algorithm initially prioritized which manual test case to select for automation. However, by exchanging test cases for scenarios and test steps for the requirements a scenario tests, the approach can be used for prioritizing which scenarios' requirements to be translated to passive test cases. The intuition is that a scenario containing requirements that occur in many other scenarios (and thus could be reused many times) should be translated early, especially if the effort of doing so is little.

The idea is to keep track of each requirement's number of remaining occurrences in the scenarios. For example, a requirement that occurs in two scenarios will have a count of two. A translated requirement is always counted as zero. This occurrence map is updated each time a new scenario shall be selected. A p/e score is calculated for each scenario as shown in Definition 1, derived from Paper A. A high p/e indicates a promising candidate to be selected.

When the most promising scenario requirements have been translated, and the corresponding logging has been adjusted, the p/e score is re-calculated for the remaining scenarios, and a new candidate scenario is selected. This update-select procedure is repeated until the set of scenarios is empty.

Translation

This section outlines the process of translating requirements into passive test cases as shown in Figure 4.16. While the details of the process presented in Paper C are still valid, this section contains a few updates. First, since the writing of Paper C, the tool support of the main definitions file has undergone substantial development. As a result, abstract signal interface definitions[7] in the requirements can reduce the concretization process to updates in the main definition files. For non-safety-critical requirements, such abstract interfaces can be constructed by successively adding the language harmonization results to the main definitions. The concretization step then connects each abstract interface signal to an expression of physical signals for a particular system version instead of updating the abstract test case. Second, new language constructs like `select` and `exist` allow expressions with optional signals. As an example from Paper E, these language constructs may increase the number of logged signals by skipping, e.g., validity- or redundant signals.

Requirement Analysis: A striking difference between traditional (active) and passive testing is that while the traditional test case controls the stimulus and thus "knows" the current status of the system, a passive test case must determine the system state without relying on the particular stimuli or the order of previous stimuli to the system. Further, while a classic test case may test the brake light by pressing the brake pedal and checking the brake light after a short delay, a passive test case must also consider a potential brake request from, e.g., another subsystem. Thus the first step in the proposed translation process includes a detailed requirements analysis. The result of the requirement analysis is knowledge about dependencies and

[7] required for safety-critical requirements by regulations such as [11, 9]

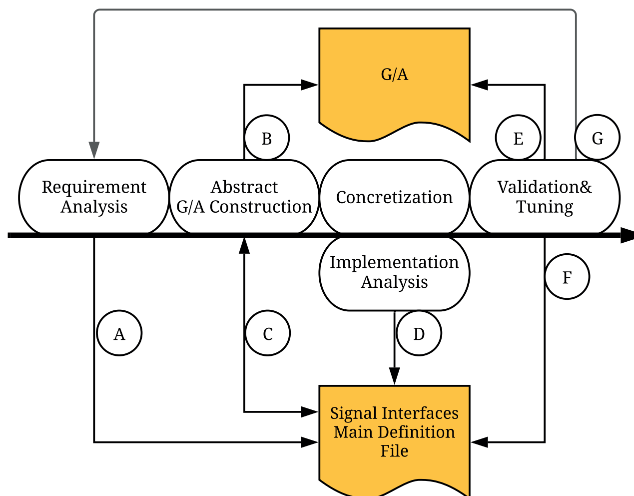


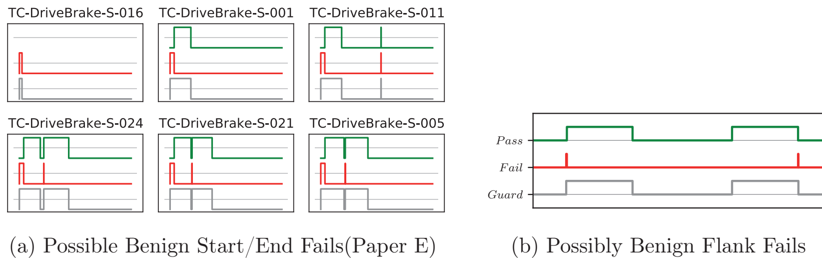
Figure 4.16: Simplified Overview of the Translation Process

other relations to the current requirement. If the requirements provide an abstract signal interface to the function, definitions are added to the main definition files as illustrated in Figure 4.16(A)

Abstract G/A Construction: Given the information gathered in the previous step, information on what to test and when is extracted. This information is matched to the already existing abstract signals or harmonized and added to the main definition file (C) in Figure 4.16. Finally, a boiler plate is selected from Listing 4.1, and the passive test case is formalized to a T-EARS expression using the identified abstract signals. This step may be repeated until all aspects of the requirement are covered.

Concretization and Implementation Analysis: Given the updates in the tool and the T-EARS language, the concretization and implementation analysis can be effectively merged. Instead of updating the passive test case, the work is to analyse the current system release and construct expressions for each abstract signal. This activity can be performed in parallel with all other activities. The result of the concretization and implementation analysis is a main definition file, allowing the passive test cases to be evaluated over a signal log from a test execution.

Validation and Tuning: While the logic of the abstract test case can be validated



(a) Possible Benign Start/End Fails(Paper E) (b) Possibly Benign Flank Fails

Figure 4.17: Fine Tuning Test Cases. Getting rid of false fails.

using hand-crafted signals during development, this step aims to make sure that the passive test case is activated when it should on actual execution log files and not reporting any false positives. In short, the working set of passive test cases are applied on a set of execution logs. In the paper, these logs came from a well-tested system, so any fails found could quickly be dismissed as false positives. In real life, however, the source of each such fail needs a thorough examination.

The tuning process starts with ruling out any system startup or shutdown disturbances. Figure 4.17a shows startup disturbances causing the test case to fail shortly at the beginning of all scenarios. The three lines (gray,red,green) in each sub-plots shows the value for the **G**uards, **P**ass and **F**ail over time. For the state patterns (BP1-3 in Listing 4.1), the tuning is focused on the guard edges as shown in Figure 4.17b. If the passive test case is passed except for a very short period at the beginning or end of a guard period, the experience from Paper E is that this is primarily due to natural latencies or sampling errors. Since the practitioners in our partner organization strive for using state patterns, due to their inherently more

extensive time coverage, not enough event patterns were used to form any conclusive advice on how to tune these.

2 RG2 - Evaluation

The second research goal (RG2) addresses the overall usefulness of the proposed methods and tools in an industrial setting. Paper A evaluates the proposed prioritization approach to test case automation candidate ordering and concludes that the algorithm is feasible to use on an industrial-sized problem. Moreover, the algorithm can be applied as suggested in Section 1.4 to prioritize requirements by scenarios instead. Figure 4.18 shows a simulation of how the number of remaining requirements would diminish over time when translating the requirements of one scenario[8] one after another in different orders. One assumption is that all requirements covered by one scenario are translated before translating the requirements of the following scenario. Another assumption is that a requirement that has been translated to a G/A can be "reused" without any further processing. Such reuse is beneficial if requirements are tested in more than one scenario.

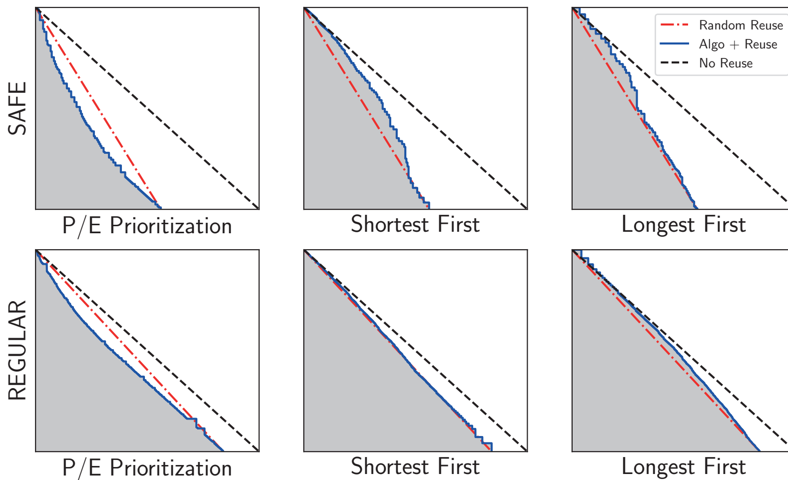


Figure 4.18: Successive Translation of the Requirements of Test Scenarios. The plots show the effect of ordering scenarios before translating their requirements to G/As using P/E Prioritization from Paper A, shortest first, and longest first. The first row concerns safety-critical requirements and the second row concerns regular (non safety-critical) requirements. The Y axis shows number of requirements remaining to translate given the scenario selection order, X axis represents the time, expressed in terms of number of translated requirements.

In Figure 4.18, the diagonal dashed line shows a comparison if all requirements are translated to guarded assertions without reusing already translated requirements.

[8]For clarity, we assume the simplification of one traditional test case testing one scenario, emphasizing the set of targeted requirements for each scenario/test case.

This line is identical for the three orders. The red dash-dotted line shows the case when the scenarios are picked by random but requirements are only translated once (This line is also the same for the three orders). Notably, this line ends later in the REGULAR case since the requirements in the SAFE case occur more often in several scenarios. The solid blue line shows the effect of the different scenario orders. When using the P/E prioritization, the SAFE case shows a quicker early reduction in the number of requirements to translate compared to the random order. A quicker reduction is beneficial if the system needs to be tested before the complete translation is finished since a greater deal of the requirements can be tested in parallel for the translated scenarios. The benefit is determined by the distance (ΔY) between the dashed or dash-dotted lines to the solid blue line at a particular point in time. The win is less for the REGULAR case, probably because the room for improvement is smaller (less reuse possible). The shortest first and longest first cases show worse performance than a completely random choice of requirements to translate.

Paper C evaluates the passive testing approach using an initial prototype of the language from [17] and concludes that the approach should be applicable for 40%-80% of the Scania test cases (at the system integration level) and that the interactivity of the tool together with the ability to load and evaluate the expressions on an actual log file, was a fundamental feature. Paper C also composes a list of proposed improvements and requirements on the tool and the language to reach industrial acceptance. The evaluation in Paper B mainly shows a proof of concept evaluation of the proposed process, core language, and updated tool-support. The evaluation continues in Paper E, where the process is concluded with fine-tuning, showing that the language, tool, and process are adequate for translating a set of requirements to passive test cases. Furthermore, after applying the complete chain of solutions in this thesis, no false positives were encountered when evaluating the passive test cases over an execution log from a test expert on an actual **H**ardware **I**n The **L**oop (HIL) test rig (Figure 4.19a). The resulting test cases were also able to find all injected

G/A	Result	Evaluation Details
REQ-244	P	9 Guard Activations,9 passes , and, 0 fails
REQ-245	P	2 Guard Activations,2 passes , and, 0 fails
REQ-246	P	2 Guard Activations,2 passes , and, 0 fails
REQ-248	P	1 Guard Activations,1 passes , and, 0 fails
REQ-253	-	"...._NoTcmsEmBr" is not logged.
REQ-254	-	"...._NoTcmsEmBr" is not logged.
REQ-255	-	"...._NoTcmsEmBr" is not logged.
REQ-258	-	"...._NoTcmsEmBr" is not logged.
REQ-259	-	"...._NoTcmsEmBr" is not logged.
REQ-260	-	"...._NoTcmsEmBr" is not logged.
REQ-281	P	6 Guard Activations,6 passes , and, 0 fails
REQ-283	-	"...._NoTcmsEmBr" is not logged.
REQ-290-A1	P	1 Guard Activations,1 passes , and, 0 fails
REQ-290-A2	P	1 Guard Activations,1 passes , and, 0 fails
REQ-349	-	"...._NoTcmsEmBr" is not logged.
REQ-350	-	"...._NoTcmsEmBr" is not logged.
REQ-456-A1	P	1 Guard Activations,1 passes , and, 0 fails
REQ-456-A2	P	1 Guard Activations,1 passes , and, 0 fails

(a) Expert no faults injected

G/A	Result	Evaluation Details
REQ-244	P	2 Guard Activations,2 passes , and, 0 fails
REQ-245	F	2 Guard Activations,3 passes , and, 1 fails
REQ-246	F	1 Guard Activations,2 passes , and, 1 fails
REQ-248	-	Guard never activated
REQ-253	-	"...._NoTcmsEmBr" is not logged.
REQ-254	-	"...._NoTcmsEmBr" is not logged.
REQ-255	-	"...._NoTcmsEmBr" is not logged.
REQ-258	-	"...._NoTcmsEmBr" is not logged.
REQ-259	-	"...._NoTcmsEmBr" is not logged.
REQ-260	-	"...._NoTcmsEmBr" is not logged.
REQ-281	F	2 Guard Activations,2 passes , and, 1 fails
REQ-283	-	"...._NoTcmsEmBr" is not logged.
REQ-290-A1	-	Guard never activated
REQ-290-A2	-	Guard never activated
REQ-349	-	"...._NoTcmsEmBr" is not logged.
REQ-350	-	"...._NoTcmsEmBr" is not logged.
REQ-456-A1	F	1 Guard Activations,2 passes , and, 1 fails
REQ-456-A2	-	Guard never activated

(b) Expert Session Finding Faults.

Figure 4.19: Final Expert Evaluation [Paper E]

faults in another expert test run, as shown in Figure 4.19b. Finally, the whole set of safety-critical requirements of TCMS was analyzed against the language presented in Paper D to see if the requirements can be expressed into passive test cases using T-EARS.

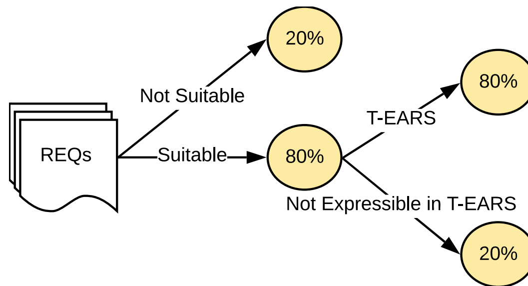


Figure 4.20: Evaluation Results Derived from Paper D. In total 64% of the safety-critical requirements were judged to be expressible in T-TEARS.

As seen in Figure 4.20, about 20% of the requirements were not suitable for any form of passive testing. Given the ones that were suitable for passive testing, a few ($\sim$ 20%) were judged not to be expressible in T-EARS, while the majority ($\sim$ 80%) were judged to be expressible in T-EARS. This corresponds to $\sim$ 64% of the complete set of requirements. This result is consistent with the Scania testers estimation in Paper C.

Chapter 5

Related Work

When deciding the point in time and which test case to automate first, there are several factors to consider and many approaches to such work-prioritization. The prioritization approach in this thesis is similar to the approach of Sabev and Grigorova [33]. Their effort estimation builds on factors identified by the authors, such as "large data inputs" and intrinsic information about the test cases. Although such factors may improve the algorithm's performance, one implicit goal of this thesis has been to avoid any factors that cannot be determined automatically from the test case itself. Further, while their model promotes the automation of test cases with high manual and low automation efforts, the proposed method in this thesis also considers the future gain of automation, given the set of already automated test cases and the remaining test cases.

One way to avoid writing the same test logic over and over again is to divide the test case into test logic and input stimuli sequence. Such separation of concerns allows reuse of the test logic independently of the input stimuli. The work presented in this thesis builds upon the concept of guarded assertions that Gustafsson et al. [21] introduced as a means of modeling passive test logic using UPPAAL[1] models. While the guarded assertion concept targets system level testing of vehicular systems, the same approach occurs in other directions of research under different names for other domains. Other works on passive testing [2, 8] uses the names *invariants* or *monitors* to denote the test logic. Although there exist some works concerning passive testing within the automotive domain [27, 34], most of the works on passive testing targets network protocols and program traces at unit or integration level while the the work in this thesis targets system level testing of signal based vehicular systems. Close to passive testing, but, with a more proactive purpose, we find Runtime Verification, that also uses the term 'monitors' for the combination of the guard and the assertion [25]. While (on-line) runtime verification strives to report a fault with the purpose of mitigating faults turning into failures, the tool in this thesis currently works off-line and off-target and thus have no intention to interact with the System Under Test. The off-line approach also allows modification and re-evaluation of test cases for the same test execution to quickly find root-causes. There are also model-based approaches such as MiLEST [38] focusing on Model-in-the-Loop (MIL)

[1]https://uppaal.org/

testing. Another concept for model-based MIL testing is Automotive Verification Functions (AFVs) [39]. It uses the terms precondition block for the guard and assertion block for the assertion and adds a final arbitration block to decide the final verdict. While this solution is currently implemented in Matlab/Simulink, the proposed specification language, T-EARS, is based on a formalization of the patterns of Easy Approach to Software Requirements Syntax (EARS) [26]. There have also been other attempts to formalize requirement patterns. Autili et al. [3] present a framework and a tool, PSPWizar, that translates requirements written in Structured English Grammar to a formal representation. Another example is the SESAMM Specifier by Filipovikj et al. [13] that simplifies the formalization of requirements into test cases using predefined patterns and Restricted English Grammar (REG). The proposed tool in this thesis differs from these tools in that the test engineer can see how the requirement is evaluated on an execution trace while the engineer is writing the requirement or test case. Further, the proposed specification language can be used for formalizing requirements using abstract signals that can be drawn in the tool as examples to the test engineer. Pange et al. [28] discuss different approaches to facilitate formalization. However, few solutions seem to have survived outside the academic context. Further, much focus is on the graphical representation of the test logic. In contrast, the proposed approach in this thesis builds on a textual model for the test logic and visualization of the evaluation instead.

Chapter 6

Conclusions

The goal of the thesis is to propose and evaluate industrially applicable methods and tools for passive testing at the system level of vehicular software systems.

Specifying such passive test cases poses challenges, such as describing a system's state and acceptable temporal variations of logical expressions. The proposed specification language (T-EARS) sacrifices detailed temporal specifications on each sub expression, only addressing the leading and trailing edges of the system response, in favor of intuitiveness. This approximation worked well for the studied systems at the system level but may not suffice at the unit level or a hard real-time system. The empirical results indicate that an intuitive language, easy-to-read, and easy-to-write, requires keeping the language small and providing structured templates or boilerplates. The overall challenge has been to balance the completeness of the language with the level of intuition. In line with other works, the provided T-EARS boilerplates, connects back to a set of standard requirements patterns (EARS).

Research on new languages for describing passive test cases requires a great deal of infrastructure. This thesis proposes a research framework consisting of open-source modules, implementing basic data integration, intuitive editing, navigation, and evaluated test case visualization. The open-source approach enables the researcher to experiment with, e.g., language constructs.

Introducing passive testing into an industrial setting requires guidance on integrating it into an existing test process. The process presented in this thesis suggests how to prioritize the requirements using a current test plan to maximize the reuse of test logic. A structured way of translating and fine-tuning test logic is also proposed.

Keeping the focus on the low to medium complex requirements allowed creating an intuitive language and IDE for passive testing feasible to test signal-based vehicular systems at the system level in an industrial context. Finally, the similarity of the proposed specification language and the requirement specification syntax, together with the possibility of creating and editing abstract signals, contribute to closing the gap between requirements specification and testing.

Chapter 7

Future Work

The empirical results suggest that passive testing gives additional requirements coverage, but to really unleash the power of passive testing, research is required on how to produce test stimuli optimized ,e.g, to cover as many G/As as possible. The proposed research framework (open-source prototype implementation of Napkin Studio) makes this possible since evaluation of G/As can be completely automated.

Although the T-EARS language is based on industrial needs in the studied case organizations, it has been kept bare minimum. While the language constructs are adequate for most of the addressed requirements, understanding how to extend the language for more challenging features such as different signal shapes and sequences without reducing the simplicity of the language poses an attractive challenge.

Further, using passive testing yields, in the optimal case, one or more guarded assertions per requirement. This increase in information artifacts and complexity necessitates further research about creating a generic mapping to the corporate information repositories and tools and processes.

Finally, while there are several benefits of evaluating passive test cases on log files from a system after the system execution, the ability to perform the evaluation in real-time as well is desirable for, e.g., exploratory testing of vehicles. This line of research brings up questions on how to use T-EARS for monitoring purposes.

Bibliography

[1] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, (2008). ISBN: 978-0-521-88038-1 (cit. on p. 5).

[2] César Andrés, Mercedes G Merayo, and Manuel Núñez. "Passive testing of timed systems". In: *International Symposium on Automated Technology for Verification and Analysis (ATVA'08)*. Berlin, Heidelberg: Springer, (2008), pp. 418–427 (cit. on p. 47).

[3] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. "Aligning qualitative, real-time, and probabilistic property specification patterns Using a structured english grammar". In: *IEEE Transactions on Software Engineering* 41.7 (2015), pp. 620–638 (cit. on pp. 3, 10, 48).

[4] Victor R. Basili. "The experimental paradigm in software engineering". In: *Experimental Software Engineering Issues: Critical Assessment and Future Directions*. Ed. by H. Dieter Rombach, Victor R. Basili, and Richard W. Selby. Berlin, Heidelberg: Springer, (1993), pp. 1–12. ISBN: 978-3-540-47903-1 (cit. on p. 16).

[5] Bart Broekman. *Testing embedded software*. Addison-Wesley, (2003). ISBN: 0321159861 (cit. on pp. 5, 6, 8).

[6] Krzysztof M Brzeziński. "Active-passive: on preconceptions of testing". In: *Journal of Telecommunications and Information Technology* (2011), pp. 63–73 (cit. on p. 3).

[7] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. "Feature interaction: a critical review and considered forecast". In: *Computer Networks* 41.1 (2003), pp. 115–141. ISSN: 1389-1286. DOI: `https://doi.org/10.1016/S1389-1286(02)00352-3` (cit. on p. 7).

[8] Ana R Cavalli, Teruo Higashino, and Manuel Núñez. "A survey on formal active and passive testing with applications to the cloud". In: *Annals of telecommunications* 3 (2015), pp. 85–93 (cit. on pp. 3, 13, 47).

[9] CENELEC. *EN 50657 Railways Applications - Rolling stock applications - Software on Board Rolling Stock*. European Committee for Electrotechnical Standardization, (2017) (cit. on pp. 10, 41).

[10] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. "Patterns in property specifications for finite-state verification". In: *International Conference on Software Engineering (ICSE'99)*. Los Angeles, California, USA: Association for Computing Machinery, (1999), pp. 411–420. ISBN: 1-58113-074-0. DOI: 10.1145/302405.302672 (cit. on p. 3).

[11] SEK Svensk Elstandard. *EN50126 - Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*. Swedish Institute for Standards, (2017) (cit. on pp. 10, 41).

[12] SEK Svensk Elstandard. *EN50129 - Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling*. Swedish Institute for Standards, (2019) (cit. on p. 10).

[13] Predrag Filipovikj, Trevor Jagerfield, Mattias Nyberg, Guillermo Rodriguez-Navas, and Cristina Seceleanu. "Integrating pattern-based formal requirements specification in an industrial tool-chain". In: *International Computer Software and Applications Conference (COMPSAC'16)*. Vol. 2. IEEE Computer Society, (2016), pp. 167–173 (cit. on pp. 3, 48).

[14] Predrag Filipovikj, Mattias Nyberg, and Guillermo Rodriguez-Navas. "Reassessing the pattern-based approach for formalizing requirements in the automotive domain". In: *International Requirements Engineering Conference (RE'14)*. Los Alamitos, CA, USA: IEEE Computer Society, (Aug. 2014), pp. 444–450. DOI: 10.1109/RE.2014.6912296 (cit. on p. 3).

[15] Daniel Flemström, Wasif Afzal, and Daniel Sundmark. "Exploring test overlap in system integration: an industrial case study". In: *Euromicro Conference on Software Engineering and Advanced Applications (SEAA'16)*. Los Alamitos, CA, USA: IEEE Computer Society, (2016), pp. 303–312. DOI: 10.1109/SEAA.2016.34 (cit. on p. 19).

[16] Daniel Flemström, Thomas Gustafsson, and Avenir Kobetski. "A case study of interactive development of passive tests". In: *International Workshop on Requirements Engineering and Testing (RET'18)*. Gothenburg, Sweden: Association for Computing Machinery, (2018), pp. 13–20. ISBN: 9781450357494. DOI: 10.1145/3195538.3195544 (cit. on pp. 22, 31).

[17] Daniel Flemström, Thomas Gustafsson, and Avenir Kobetski. "SAGA toolbox: interactive testing of guarded assertions". In: *International Conference on Software Testing, Verification and Validation(ICST'17)*. IEEE Computer Society, (2017), pp. 516–523 (cit. on pp. 3, 19, 23, 44).

[18] Daniel Flemström, Thomas Gustafsson, Avenir Kobetski, and Daniel Sundmark. "A Research roadmap for test design in automated integration testing of vehicular systems". In: *International Conference on Fundamentals and Advances in Software Systems Integration (FASSI'16)*. (2016) (cit. on p. 3).

[19] Kevin Forsberg and Harold Mooz. "The relationship of system engineering to the project cycle". In: *INCOSE International Symposium*. Vol. 1. 1. Wiley Online Library, (1991), pp. 57–65 (cit. on p. 6).

[20] Tony Gorschek, Per Garre, Stig Larsson, and Claes Wohlin. “A model for technology transfer in practice”. In: *IEEE Software* 23.6 (2006), pp. 88–95 (cit. on pp. 16, 18).

[21] Thomas Gustafsson, Mats Skoglund, Avenir Kobetski, and Daniel Sundmark. “Automotive system testing by independent guarded assertions”. In: *International Conference on Software Testing, Verification and Validation Workshops (ICSTW’15)*. IEEE Computer Society, (2015), pp. 1–7. DOI: `10.1109/ICSTW.2015.7107474` (cit. on pp. 3, 13, 22, 23, 47).

[22] Oliver Hoehne. “The SoS-VEE model: mastering the socio-technical aspects and complexity of systems of systems engineering (SoSE)”. In: *INCOSE International Symposium*. Vol. 26. 1. Wiley Online Library, (2016), pp. 1494–1508 (cit. on p. 6).

[23] ISO/IEC/IEEE. *Software and systems engineering — Software testing - Part 1: Concepts and definitions*. International Organization for Standardization, International Electrotechnical Commission, (2013) (cit. on p. 10).

[24] Barbara Kitchenham and Stuart Charters. *Guidelines for performing systematic literature reviews in software engineering*. Tech. rep. EBSE 2007-001. Keele University and Durham University Joint Report, 2007. URL: `http://www.dur.ac.uk/ebse/resources/Systematic-reviews-5-8.pdf` (cit. on p. 18).

[25] Martin Leucker and Christian Schallhart. “A brief account of runtime verification”. In: *The Journal of Logic and Algebraic Programming* 78.5 (2009), pp. 293–303 (cit. on p. 47).

[26] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. “Easy approach to requirements syntax (EARS)”. In: *International Requirements Engineering Conference (RE’09)*. IEEE Computer Society, (2009), pp. 317–322 (cit. on pp. 10, 23, 48).

[27] Pramila Mouttappa, Stephane Maag, and Ana Cavalli. “Monitoring based on IOSTS for testing functional and security properties: application to an automotive case study”. In: *Computer Software and Applications Conference (COMPSAC’13)*. IEEE Computer Society, (2013), pp. 1–10. DOI: `10.1109/COMPSAC.2013.5` (cit. on p. 47).

[28] Cheng Pang, Antti Pakonen, Igor Buzhinsky, and Valeriy Vyatkin. “A study on user-friendly formal specification languages for requirements formalization”. In: *Industrial Informatics (INDIN’16)*. IEEE Computer Society, (2016), pp. 676–682 (cit. on p. 48).

[29] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. “Systematic mapping studies in software engineering”. In: *International Conference on Evaluation and Assessment in Software Engineering (EASE’08)*. Italy: BCS Learning & Development Ltd., (2008), pp. 68–77 (cit. on p. 18).

[30] Mauro Pezzè and Michal Young. *Software testing and analysis: process, principles and techniques*. Wiley & Sons Inc., (2007). ISBN: 13978-0-471-45593-6 (cit. on pp. 5–7).

[31] Guillermo Rodriguez-Navas, Avenir Kobetski, Daniel Sundmark, and Thomas Gustafsson. "Offline analysis of independent guarded assertions in automotive integration Testing". In: *International Conference on Embedded Software and Systems (ICESS'15)*. IEEE Computer Society, (2015), pp. 1066–1073. DOI: 10.1109/HPCC-CSS-ICESS.2015.251 (cit. on pp. 3, 13, 23).

[32] Per Runeson and Emelie Engström. "Software Product Line Testing – A 3D Regression Testing Problem". In: *International Conference on Software Testing, Verification and Validation (ICST'12)*. Los Alamitos, CA, USA: IEEE Computer Society, (Apr. 2012), pp. 742–746. DOI: 10.1109/ICST.2012.167 (cit. on p. 18).

[33] Peter Sabev and Katalina Grigorova. "Manual to automated testing: An effort-based approach for determining the Priority of Software Test Automation". In: *International Journal of Computer, Electrical, Automation, Control and Information Engineering* 9.12 (2015), pp. 2123–2129 (cit. on p. 47).

[34] Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, and Radu Grosu. "Applying runtime monitoring for automotive electronic development". In: *International Conference on Runtime Verification (RV'16)*. Berlin, Heidelberg: Springer, (2016), pp. 462–469 (cit. on p. 47).

[35] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to advanced empirical software engineering*. Springer, (2008) (cit. on p. 18).

[36] Wei-Tek Tsai, Xiaoying Bai, Richard Paul, Weiguang Shao, and Vijyant Agarwal. "End-to-end integration testing design". In: *International Computer Software and Applications Conference (COMPSAC'01)*. Los Alamitos, CA, USA: IEEE Computer Society, (Oct. 2001), p. 166. DOI: 10.1109/CMPSAC.2001.960613 (cit. on p. 7).

[37] Claes Wohlin. "Guidelines for snowballing in systematic literature studies and a replication in software engineering". In: *International Conference on Evaluation and Assessment in Software Engineering (EASE'14)*. London, England, United Kingdom: Association for Computing Machinery, (2014). ISBN: 9781450324762. DOI: 10.1145/2601248.2601268 (cit. on p. 18).

[38] Justyna Zander-Nowicka. *Model-based testing of real-time embedded systems in the automotive domain*. (2008) (cit. on p. 47).

[39] Justyna Zander-Nowicka, Ina Schieferdecker, and Abel Marrero Perez. "Automotive validation functions for on-line test evaluation of hybrid real-time systems". In: *Autotestcon*. IEEE Computer Society, (2006), pp. 799–805. DOI: 10.1109/AUTEST.2006.283767 (cit. on p. 48).