Industrial Scale Passive Testing with T-EARS

Daniel Flemström
Mälardalen University
Sweden
daniel.flemstrom@mdh.se

Henrik Jonsson
Bombardier Transportation AB
Mälardalen University
Sweden
henrik.x.jonsson@mdh.se

Eduard Paul Enoiu
Mälardalen University
Sweden
eduard.paul.enoiu@mdh.se

Wasif Afzal
Mälardalen University
Sweden
wasif.afzal@mdh.se

***Abstract*—**
Passive testing continuously observes the system or system execution logs without any interference or instrumentation to test diverse combinations of functions, resulting in a more thorough evaluation over time. However, reaching a working solution to passive testing is not without challenges. While there have been some efforts to extract information from system requirements to create passive test cases, to our knowledge, no such efforts are mature enough to be applied in a real, industrial safety-critical context. Our passive testing approach uses the Timed - Easy Approach to Requirements Syntax (T-EARS) specification language and its accompanying tool-chain. This study reports challenges and solutions to introducing system-level passive testing for a vehicular safety-critical system through industrial data analysis, including 116 safety-related requirements. Our results show that passive testing using the T-EARS language and its tool-chain can be used for system-level testing in an industrial setting for 64% of the studied requirements. We identified several sources of false positive results and show how to tune test cases to reduce such false positives systematically. Finally, we show the requirement coverage achieved by a manual test session and that passive testing using T-EARS can find a set of injected faults that are considered hard to find with other test techniques.

***Index Terms*—software testing; passive testing; test case automation;**

I. Introduction

Faulty software in safety-critical systems may cause economic damage and, in some cases, loss of human lives. At the same time, the embedded software in today's safety-critical systems grows more and more complex, which emphasizes the importance of efficient and realistic testing. One of the methods suggested allowing that is passive testing [1]. The idea is to only *observe* the system under test, which allows, e.g., parallel execution of test cases. In this approach, test cases only monitor the system under test (SUT) and do not alter the state of the system at all. Instead, requirements are verified whenever appropriate, independently of the input stimuli sequence. Cavalli et al. [2] give a more general in-depth review of passive testing methods in various domains related to networked systems. Notably, most of these approaches target non-critical software testing, such as protocol testing in, e.g., web applications and telecom applications. Such applications often involve sending and receiving complex data. While the current passive testing methods have a long history for these applications, the safety-critical software we address does not require handling comprehensive data transfers and complex state machines. Further, due to the complexity of the data and the state machines in the original domain, the specification languages used are very complex and often based on mathematical expressions. While it has been successfully used, e.g., in protocol testing [2], its practical use has been limited by its reliance on formal specifications [3], [4], [5].

Although there have been attempts to introduce passive testing to the system-level testing of safety-critical systems [6], [7], [8], [9], these studies are all somewhat limited when it comes to industrial validation and adoption. Also, these do not consider practical issues and tooling requirements.

Gustafsson et al. [7] previously introduced the concept of independent guarded assertions (G/As) as an approach for passive testing at the system level in the vehicular software domain. As in contemporary passive testing, the idea is to treat the input stimuli (that affects the system state) and the test oracle (i.e., that decides if a system requirement is fulfilled or not) independently. An G/A describes such an oracle by defining a *guard*, G, that decides whether one or more *assertions*, A, are expected to be fulfilled or not. Given some of the recent advances, the question we ask is if passive testing now mature enough for industrial adoption? Using our previous work with the T-EARS (Timed - Easy Approach to Requirements Syntax) Specification language and the SAGA (Situation Aware Guarded Assertions) tool-chain ([10], [11]), this paper presents a new study of introducing passive testing at the system level in an industrial safety-related vehicular software system. The issues and benefits encountered during practical deployment of passive testing are of particular interest.

The main contributions of this paper are:

- An industrial case study of introducing passive testing at the system level showing how this technique can be adopted and deployed in the safety-critical domain,
- Experiences in using the T-EARS language with its accompanying tool-set as a means of expressing concrete passive test cases running at the system level,
- Challenges concerning industrial adoption maturity for writing passive test cases, and
- A systematic process of tuning passive test cases concerning false positive outcomes.

The rest of the paper is organized as follows. In Section II, we describe the T-EARS language and its tool-chain for passive testing. In Section III we outline the method used in this industrial case study while in Section IV we report the results before we outline the related work in Section V and conclude this paper in Section VI.

II. Background to T-EARS and its Tool chain

T-EARS, short for, Timed - Easy Approach to Requirements Syntax [11] was introduced in 2017 as an easy to use specification language for expressing passive test cases and has been continuously improved since then. The basic structure of the language and the goal of simplicity are inherited from EARS (Easy Approach to Requirements Syntax) [12].

```
'Bp-1*' = while true shall I
'Bp-2*' = while I1   shall I2
'Bp-3'  = while I1   shall I2 within t
'Bp-4*' = when P shall I
'Bp-4'  = when P1 shall P2 within tw
'Bp-6'  = when P shall I for tf within tw
'Bp-7'  = when P shall I within tw for tf
```

Listing 1: T-EARS Boiler Plates (P = `Events`, I = `Intervals`, and t, tw, tf denotes a time specification, e.g., 250ms

T-EARS relies on five cornerstones: *Boilerplates, Data Types, Expressions*, *Timing* and *Structural Elements*. The boilerplates are shown in Listing 1 and describe different ways to form a passive test case (G/A) in T-EARS. Their purpose is to cover the most commonly occurring test case structures rather than being canonical, e.g., BP-1 and BP-2 are indeed special-cases of BP-3 if only considering the grammar, but yet, representing different requirement constructs. The second cornerstone is the Data Type. While EARS requirements are expressed in natural language, corresponding G/As need to use machine-readable entities. Examples of data types are `Signal`, `Intervals` and `Events`. The data types allow the third cornerstone, Expressions. Expressions combine data types using Boolean and mathematical operators as well as functions such as absolute values, bitmasking, and edge detection. The fourth cornerstone is Timing and allows e.g. accepting a delayed response, specifying required length of a response or filter intervals based on length in time. Finally, the last cornerstone *Structural Elements* allows hiding complex expressions and map abstract signals to the implementation using keywords like `def, const`, and `alias`.

Consider the following illustrative example of a safety-critical requirement: *"when doors are open, traction shall be disabled"* and its representation in T-EARS, shown in Listing 2.

```
1 while Doors_are_open == true
2 shall
3     Traction_enabled == false within 300ms
```

Listing 2: Example Guarded Assertion

```
1 alias Traction_enabled =
2    MWT.xxx.yyy,EnTrMio
3 def intervals Doors_are_open =
4     MWT.xxxxx.yyyy.LtfDrCldLck == false and
5     MWT.xxxxx.yyyy.RgtDrCldLck == false
```

Listing 3: Signal Mapping Example

The guard expression at line 1 in Listing 2 decides whether the doors are open or not (a sequence of time intervals where the guard is true), and the assertion expression at line 3 would evaluate to true whenever traction is allowed. For each guard interval, as long as the assertion expression evaluates to true, the test is considered to be passed. Conversely, if the assertion expression evaluates to false any time during the guard interval, the test has failed during those intervals. In a large (often distributed) system, signals are naturally delayed. Fail intervals due to such delays can be masked out using the `within` clause on line 3, allowing a delayed response of max 300ms. Outside the guard intervals, the result of the assertion expression is not evaluated. Using the above example on a real log requires mapping the logical signals (e.g., `Doors_are_open`) to real (technical) signals in the log file. Such mapping is done in a definition file, as shown in Listing 3. The first line shows a trivial `alias` that can be defined at any number of levels. The second line shows a logical signal that requires two technical signals from the implementation.

The T-EARS tool-chain was first introduced in [11] as the *SAGA tool*. It has, together with the language, been continuously updated after an industrial evaluation in [13]. The tool's purpose is to support the entire process of writing, tuning, and finally evaluating passive test cases. The key features are:

- **Signal Exploration**- The tool allows logs of different formats to be loaded one by one or merged for comparison. The logged signals can be viewed, zoomed, and panned individually or in groups using a synchronized signal plot view.
- **Interactive Text Editing** - A text editor supporting code completion and abstract signal lookup is provided. A key feature is the ability to evaluate the current expression on a loaded log file interactively. Further, sub-expressions are shown as signal plots beneath the editor.
- **Interactive Signal Editing** - The ability to create editable abstract signals as well as editing existing signals is key when debugging complex expressions. The evaluation of the current expression is interactively updated when editing signals. The evaluation result is then overlaid on all plots.
- **Batch Evaluation and Tuning** - The evaluation core is capable of evaluating sets of passive test cases over sets of log files as well as creating overviews that allow each evaluation (with the corresponding files) to be opened in the SAGA Tool with a mouse click.

The process of using the language and the tool to translate natural language requirements to G/As was outlined in [10], having the main activities:

- *Requirement analysis:* Identify guard and assertion expressions using logical signals and values.
- *Abstract G/A construction:* Construct abstract G/As through high-level (logical) signal and values representing states and events.
- *Implementation analysis:* Identify the design documentation to resolve abstract signals and values to concrete.
- *Concretization:* Refine constructed G/As and signal mapping to be evaluated on system logs.
- *Tuning & validation:* Examine and fine-tune the guard and assertion expressions.

We follow these steps in this paper to show the encountered challenges and the solutions used to make passive testing applicable in industrial-scale safety-critical system development.

III. Method

In this section, we outline the case study we performed in an industrial setting, following [14].

A. Study Objective

The objective is to study the practical implications of adopting passive testing using T-EARS and the accompanying tool-chain to a safety grade industrial context. In particular, we analyze the challenges encountered and our solutions to those challenges when writing, executing, and analyzing passive tests written in T-EARS and its accompanying tool-chain. The study's context is the *system level* testing in an embedded vehicular software system.

B. Case organization & Unit of Analysis

The studied case organization is responsible for developing TCMS (Train Control and Management System), an embedded safety-critical system controlling and monitoring software and hardware systems in a train. The different functions of a TCMS system constitute safety-related parts (SAFE), controlling safety-related functions, and non-safety-related parts (REGULAR) for non-safe control and monitoring functions written using the IEC 61131-3 programming language [15]. The focus of this study is on the safety-critical part (SAFE) of TCMS. The engineering processes of TCMS software development are performed according to safety standards and regulations (e.g., EN 50128 [16]). Testing can be performed on a real train or different configurations of a simulated virtual train environment, illustrated in Figure 1. The "Train" includes everything required to perform end-to-end testing of the system under test (SUT). Typically either simulated systems and train environment or a hardware (HIL-Rig) are used. In either case, the automated and manual test cases are not changed. Such tests can either be performed by a tester (in manual testing) or implemented as automated test scripts (performing the same sequence of actions and reaction checks as the manual tester would have done). Both manual and automated test cases can be used to log signals using the same logger, which is vital for applying a passive testing approach.

The unit of analysis is the set of safety-related requirements, implementation and system-level tests for a Safety Integrity Level 2 (SIL2)[1] compliant TCMS application.

C. Safety Related Requirements

The system-level test cases are primarily written in natural language for manual test execution. These must be carefully written and reviewed to ensure that they cover all requirements and combinations and are feasible for testing in the intended test environment (i.e., the actual train). These tests take hours of manual labor to complete. However, while these manual tests provide valuable information about irregularities and contradicting results that would be hard to observe by strictly automated test cases, intermittent failures in unexpected situations are still difficult to catch with such a traditional testing approach.

[1]EN 50126 - Railway applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)

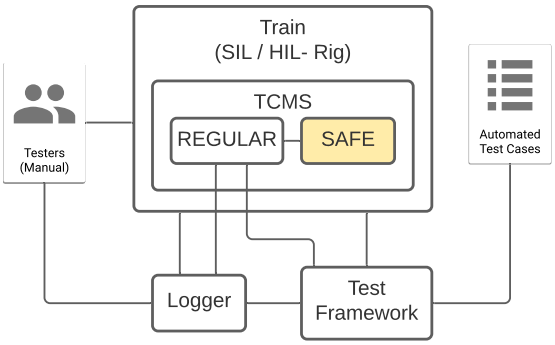

Fig. 1: Conceptual Overview, TCMS Testing Environment

As long as any cab door is not closed and locked, start inhibit shall be set.
INPUT:
- (MIO-S): Cab doors closed and locked in cab x = false or invalid (both redundant signals from any cab in train)

OUTPUT:
- (IP) 'Start inhibit reason' includes 'Cab doors not closed'
- (internal): Start inhibit = true

Fig. 2: Example Requirement (slightly adjusted for readability)

The studied requirements are described in a semi-formal format, as shown in Figure 2. Since there are two cab compartments for the train driver, A1 and A2, these are typically referred to as Cab x, meaning that the requirement applies to both Cab A1 and Cab A2. The example in Figure 2 starts with a natural language description of the requirements. The SAFE requirements have an additional semi-formal description, starting with an INPUT section, describing the condition under which the requirement shall be fulfilled. The condition includes a list of logical input signals, their respective values, and logical relations. In this example, the means of communication (e.g., MIO-S or IP) is also given to provide a bridge to the concrete system. This structure corresponds quite well to a *guard* in the G/A concept. The OUTPUT section lists logical signals and expected values (response) to the INPUT section's conditions. This part corresponds well to an assertion in the G/A concept.

Safety-critical requirements typically summarize these logical INPUT and OUTPUT signals, so the same names are used throughout an entire function or system.

D. Case Study Procedure

The study is carried out in three phases. In the *first* phase, a gold standard is set, and a set of requirements are chosen. This gold standard is used for evaluating a successful translation of the requirements to passive tests (G/As) concerning false positives. The absence of such false positives is the stopping criterion for the *second* phase, the adoption of the translation process [10] using T-EARS and its accompanying tool-chain. Although the core of this paper refers to the implications of adopting and deploying passive testing in practice, the *third* phase is an evaluation that the results are applicable, the produced G/As do not elicit false positives, and are capable of finding faults that are hard to find using the existing tests.

1) Phase 1, Gold Standard and Requirements Selection: The first phase, setting the gold standard, requires selecting a set of requirements to translate and a set of existing test cases testing those requirements. The following steps are followed:

- First, the TCMS system requirements are analyzed to find automated test cases and, ultimately, a set of log files, serving as the gold standard. The chosen test cases shall stem from a system that is well tested, sufficiently observable, and also offer reliable automated test scripts. Automation is key here since it should be possible to re-execute the test cases with any extra logging required by the G/As.
- Second, a set of requirements is selected where the selected automated test cases test each requirement, and dependent signals are observable from the selected set of test cases.

This phase's outcome is a set of automated test cases and a set of requirements to be translated into passive test cases (G/As).

2) Phase II, Process Adoption: This phase's starting point is the T-EARS language and its accompanying tool-chain, together with the proposed process of translating natural language requirements to passive test cases as G/As [10]. The purpose of this step is to explore how to apply this in practice in an industrial safety-critical context.

Inputs to this phase are a set of automated test cases for a set of well-tested system functions and a set of requirements tested by those test cases. During the adoption procedure iterations, requirements are translated using the translation process [10] and compared to the gold standard. Iterations could cover the entire process from the requirement to tuned G/A or a particular step in the process, meeting a particular challenge. This iterative adoption procedure is based on the work of Staron [17]. Each iteration contains the following activities:

- *Diagnosing*: Observations and analysis of challenges met during the different steps of the translation process.
- *Action-Planning*: Discussions with other industry experts on possible solutions.
- *Action-Taking*: Implementation or stubbing of suggested features.
- *Evaluation*: Occurs at each iteration that ends with a set of tuned G/As. The resulting G/As are evaluated against the set of correct logs.
- *Learning*: Information and reflections on the work performed are collected. The notes are then analyzed and structured into a process, and tests are executed.

This phase's result is a set of improvements to the original process, a set of challenges that could not be solved or requirements on new tools, and a set of tuned G/As, translated from the requirements using the improved process.

3) Phase III, Final Evaluation: The last phase in the case study is the final evaluation.

- The complete set of 116 SAFE requirements is analyzed given the improvements from phase two. The suggestions are implemented to see the extent to which the final results are useful.
- Further, the resulting G/As from the second phase are evaluated over two signal logs from a manual test session with an expert.

The results of this phase are a measurement of false positives from a well-tested system, as well as means to assess the degree to which the produced G/As can find injected faults that are hard to find using existing test cases.

IV. Results and Discussion

The results of the case study's three phases are outlined as follows: Section IV-A presents the Phase 1 - *Gold Standard and Requirements Selection*, Phase II - *Process Adoption* is presented in the Sections IV-B to IV-F, and finally, Phase III *Final Evaluation* is presented in the Section IV-G.

A. Phase I - Gold Standard and Requirements Selection

The first phase aims to identify a set of automated test cases to use as a reference gold standard when translating a subset of the requirements to G/As and a set of requirements to translate. We identified 14 automated regression test cases from the Drive-and-Brake Functions, fulfilling our criteria on automation, observability, and priority by the case organization. The tested requirements were selected for translation, while the final evaluation is done over 116 SAFE requirements from the overall TCMS system.

As described in Section III, the translation process [10] we earlier outlined is used as a starting point and framework for structuring the results. Further, since one of the potential drawbacks of the method is the risk of false positives [13], we use a subset of the regression test log files from a well-tested system as a gold-standard. Sixteen of the tested requirements were translated and tuned until all false positives were removed, and no remaining problems were left unresolved. We then analyzed each challenge and possible solution and formed a generalized workflow.

B. Phase II - Requirement Analysis Results

Given the choice of the Drive-and-Brake Functions in Section IV-A, the expected result of this activity is the set of

requirements concerning the Drive-and-Brake Functions, dependencies, and a list of logical signals. Typically, these logical signals result from the harmonization activity [10]. However, using logical signals already in the requirements is common in safety-related requirements. Such standardization radically reduced effort in the later steps of the translation process. We argue that writing requirements this way is worthwhile on non-safe requirements as well. Besides speeding up the translation work, using logical signals disconnects the passive test cases from a particular release of the system, which was imperative since signals tend to be frequently reallocated between data buses or modules, especially in early releases. Finally, since the list of used signals is known at an early stage, the implementation analysis can start in parallel, so logging of the required signals can be done early, which ultimately reduces the time spent in the concretization step (Section IV-E). The ability to transparently observe this mapping while translating the requirement was beneficial since the test engineer is more confident in the technical signals' meaning.

For requirements that contain more complex expressions such as sequence, many signals, or negated logical expressions, we can manually create examples signals to facilitate the next step, namely the Abstract G/A Construction. Due to observability limitations in the testing framework available to us, we selected 16 of the identified Drive-and-Brake requirements for translation. In parallel with this step, parts of the *Implementation Analysis* (as described in Section IV-D) and *Concretization* (as described in Section IV-E) were performed. In short, the required technical signals were identified and added to the regression test cases to create the gold standard log files. Further, the mapping from logical to technical signals was prepared.

Throughout the upcoming sections, we use these logs to show the progression from a drafted G/A to a tuned G/A with a minimum of false positives.

C. Phase II - Abstract G/A Construction Results

The expected outcome from this step is a G/A that is complete with respect to its logical guard and assertion expressions, but using logical signals. The main steps in this activity are [10] *a) Language Harmonization*, *b) Extraction Of G/A information*, *c) Pattern Selection*, and *d) Abstract G/A Formalization*. Since (a) and (b) steps are already given by the semi-formal notation of the requirement, we focus on (c) and (d) steps.

One observed challenge, also shown in Figure 2, is the case when there are alternative states for the guard (from any cab in the train), which in this case is cab A1 or cab A2. We observed that, when possible, splitting the passive test into one G/A for each OR expression allows a more fine-grained test, e.g., allowing each cab to be tested separately. Further, when the validity of a signal S (i.e., `S and valid_S`) is important to test, (`S or not(valid_S)`) could be used to create two tests as previously mentioned. The use of standardized logical signals from the previous step allows a straight forward translation of the requirement shown in Figure 2.

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	F	F	F	F	F	F	F	F	F	F	F	[F]	F	F
REQ-283	F	F	F	F	F	F	F	F	F	[F]	F	F	F	F
REQ-290-A1	F	F	F	F	F	F	F	F	F	F	[F]	F	F	F
REQ-290-A2	F	F	F	F	F	F	F	F	F	F	[F]	F	F	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	-	-	-	-	-	-	-	-	-	-	-	-	-	[-]

Fig. 3: Initial Translation evaluated over a set of passed logs

Listing 4 shows how the G/A at Line 2-7 can be split into one G/A for each cab. The example requirement did not contain any timing information. We note here that it is not always possible to have precise timing information for each requirement at this level. Often such timing behavior is described in separate non-functional requirements covering one or more functional requirements. Even if a requirement contains sufficient timing information, valuable information can be obtained by observing the results of a G/A without timing specifications added.

```
1  // Short but does not fail for particular cab
2  'REQ-246' = while
3    MIOS_IN_Cab_drs_closed_and_locked_cab_A1 == false
4    or
5    MIOS_IN_Cab_drs_closed_and_locked_cab_A2 == false
6  shall
7      start_inhibit_reason_includes_cab_door_open
8  // Split G/A may give more details on fails.
9  'REQ-246-A1' = while
10   MIOS_IN_Cab_drs_closed_and_locked_cab_A1 == false
11 shall
12     start_inhibit_reason_includes_cab_door_open
13 'REQ-246-A2' = while
14   MIOS_IN_Cab_drs_closed_and_locked_cab_A2 == false
15 shall
16     start_inhibit_reason_includes_cab_door_open
```

Listing 4: Abstract G/A

Each requirement was translated to one or more G/As using the requirements' INPUT/OUTPUT sections and the list of logical signals. No timing information was specified yet. Where necessary, we will add timing to the G/As during the upcoming tuning session. The G/As were evaluated on the set of gold standard log files, and the result is presented in Figure 3. The rows show the G/As are named according to the requirement (e.g., REQ-245) to maintain traceability. Some requirements result in more than one G/A. We use a suffix for those G/As (e.g., -A1, -A2 to denote testing Cab A1 and Cab A2, respectively). For each TC-G/A combination, a **P**, **F**, or **-**

denotes passed[2], failed or not-activated, respectively. For each log, the tested requirements (G/As) are marked with a bracket **[ ]**. For example, TC-001 tests requirement number 259. The corresponding G/A REQ-259 is thus expected to be activated (and passed). Since all logs stem from passed regression test cases of a well-tested system, we expect all such cells with a bracket to carry a **[P]**. However, without any adaptation, it turned out that a) conditions for all expected G/A were not present in the test data set ([-]) and b) several G/A failed, although the test data was expected to show passed ([F]), and c) some G/A failed in all test runs (F). In our case, any fails or ([-]) are false positives since we use reliable logs from a well-tested system. The systematic work of analyzing the cause and eliminating these false positives is described in Section IV-F.

D. Phase II - Implementation Analysis Results

This step's expected outcome is a mapping between the abstract (logical) signals and the concrete (technical) signals that can be used for directly evaluating the abstract G/As. We performed this step concurrently with the requirement analysis step in Section IV-B to facilitate the translation in the previous section.

This step's most significant challenges concern consistency, technical signal identification, observability, maintenance, and signal scoping. In an initial attempt, as a preparation for this case study using other requirements without standardized logical signals, the mapping *consistency* quickly eroded into multiple/duplicate definitions and resulted in constant updates of the automated test cases to obtain the required signals logged. *Identifying* the correct signal among the tens of thousands available was another challenge aside from the experienced observability issues. Some requirements include non-observable internal signals. Often, other signals could be used as a proxy. The challenge is to understand how the non-observable signal affects the observable signal. The next challenge is *maintaining the mappings* when the implementation changes, such as an updated source, emitting the signal. Lastly, there are always more signals available in the general case than it is possible to log. For a practical application of passive testing, this has a significant impact on the method's usefulness. If a single signal is missing, a whole passive test case is rendered useless and will not even evaluate, resulting in a potential false positive. Further, for each set of signals to log, all the test cases need to be executed. Keeping track of the log-sets by hand was tedious and error-prone.

In the upcoming paragraphs, we present identified solutions to these challenges. Firstly, having a standardized set of logical signals used for all (at least a relevant subset) requirements is critical for *consistency*. Listing 5 shows examples of the outcome of this step using this strategy.

[2]Since a G/A may be activated several times during a log, the final result can be a number fails and passes. We consider pass if no 'fails' occur, and at least one pass.

```
1  // Separating from implementation / versions
2  alias S_DrEnRgtTrLn_P =
3        MWT.xxxxx.C2M22m1In3_S_DrEnRgtTrLn_P
4  alias V_DrEnRgtTrLn_P =
5        MWT.xxxxx.C2M22m1In3_V_DrEnRgtTrLn_P
6  // Abstracting away validity handling
7  def intervals DrEnRgtTrLn_P =
8      S_DrEnRgtTrLn_P == true
9      and valid_DrEnRgtTrLn_P
10 // ...and so on
11 // Redundant validated signals using above aliases
12 def intervals
       MIO_S_Safe_door_enable_right_TRUE_and_VALID =
13   (S_DrEnRgtTrLn_P == true and
14    V_DrEnRgtTrLn_P == true) or
15   (S_DrEnRgtTrLn_R == true and
16    V_DrEnRgtTrLn_R == true)
```

Listing 5: Partially Obfuscated Signal Definitions

Further, Lines 2-3 and 4-5 show a low-level separation of logical signals and their binding to a particular implementation. Such low-level separation allows for automating the logical to technical signal mapping, contributing to solving the *maintenance challenge*. In the example, the logical signals gradually increase the abstraction level (Line 7 and finally, Lines 12-16). The "final" signal on Line 12-16 is the "standardized" logical signal used. Theoretically, it would be possible to use the logical names in the INPUT/OUTPUT section as-is, but the presence of white-space and the slight variations (e.g., CCU IP OUT / OUT CCU IP) are deemed to produce hard to find errors. Hence, the T-EARS name was created by replacing white-space with an underscore, reformatting the IN/OUT and BUS info to the same order everywhere. While translating, the tester copy/pastes the logical signal's natural language name into a search view and gets the closest matching T-EARS names. The above method had a substantial impact on the translation effort.

In the case study, we identified two approaches for increasing the logging-efficiency and possibly reducing the *scoping challenge*. The first approach concerns telegrams where each bit corresponds to a digital signal. Depending on the logging framework, logging the telegram rather than the individual signals may drastically increase the number of logged signals. If the logged telegram is a 16-bit integer, we can log the entire telegram and let T-EARS mask out the individual signals using the `bitmask` function. Such a mapping can typically be automated using a template, as shown in Listing 6.

```
1  // Logging several signals in same telegram
2  // Bitmask out individual signals
3  def interval my_bin_signal =
4     bitmask(TelegramXYZ, my_bin_sig_mask) ==
          my_bin_sig_mask
5  // Defining an optional VALID signal
6  def intervals valid_DrEnRgtTrLn_P =
7        select(exists(Valid_DrEnRgtTrLn_P)
8                      Valid_DrEnRgtTrLn_P,
9                      true)
10 def interval DrEnRgtTrLn_P =
11         S_DrEnRgtTrLn_P and
12         valid_DrEnRgtTrLn_P
```

Listing 6: Tricks For Increasing Log Information

Using the T-EARS `select` and `exists`, the validity signal of fail-safe signals can be made optional, as shown in Line 6-9 in Listing 6. The defined signal is equal to the validity

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]

Fig. 4: Activation and Startup (`ignore`) Tuned

signal if it is logged or always true if it is not logged. Using optional validity signals only works on fail-safe signals with a guaranteed fail state whenever the validity signal is false.

E. Phase II - Concretization Results

The concretization step's expected outcome is a G/A evaluated using concrete (technical) signals. Given our suggestions for the implementation step, this is accomplished by specifying the signal mapping described in the previous section. We did not observe any particular challenges in this step.

F. Phase II - Tuning and Validation Results

This step's expected result is an executable and complete G/A with a minimum of false positives. A G/A is considered complete when there are no unknown dependencies that would cause a wrong verdict. A fictitious example is a G/A only checking that the brake light is turned off when lifting the brake pedal, but another subsystem issues a brake order that causes the test to fail. False positives include such conditions and G/As are not being activated when supposed to or failing where there is no underlying fault. In this step, the tester logs required signals while operating the system to activate the G/A. Eliminating false positives turned out to be the most challenging part of the entire process. To find out how to do this systematically, we tuned our G/As against our gold standard until no more false positives were encountered. The result is presented as a systematic process containing the steps outlined in the following sub-sections.

1) Validating Guard Activation: The expected outcome of this step is a guard that is activated when expected and also not activated when not expected. In the general case, the tester operates the system (while logging the appropriate signals) until she knows that the system state is correct. For our gold standard, we already know which G/A should be active for each log. Thus, this step's expected outcome is that each G/A is activated at least once for the logs testing the G/As requirement. Such expected activations are marked with brackets in Figure 3. G/A activations for other logs are not necessary but welcome, as long as they do not produce false positives. Notably the REQ-456-A2 in Figure 3 has not been activated when expected.

We identified possible sources of missing activation(s), including input stimuli sequence, log, signal mapping, and, requirement. The first source, input stimuli sequence, is the easiest to investigate. Examining the test case actions should reveal if the test does not put the system in a testable state (as it should have). In our case, we know that the logs are correct, so the cause of the missing activation of TC-068 and REQ-456-A2 in Figure 3 must be related to some other reason. If the evaluation completely fails for one log but looks fine for another, the log probably lacks one or more signals. Another reason for not activating (or giving a faulty evaluation) may be that a signal is captured with the right name but with the wrong values. This may happen if the test framework injects faults or alter signals for testing purposes. Another common mistake is that a logical signal is defined with the wrong technical signal (e.g., a logical signal from cab A1 and cab A2 are mapped to the same technical signal by mistake). In this study, we experienced all of the above, mostly due to a lack of proper automation of these tasks. The last category, requirement, includes reasons such as the requirement lack of information to capture the testable state, or the requirement text is misunderstood. It may also be missing signals or unknown dependencies as in the example of the brake pedal in Section IV-F. In our example, it turned out that REQ-456-A2 suffered from misunderstandings of the requirement text. Updating the conditions of the guards activated the G/A for TC-017 and TC-068, as seen in Figure 4. This also shows that the G/A still fails for TC-068 and more tuning is required for the G/A.

2) Systematic Issues: This step is applicable if there exists a set of logs and a suspicion that the false positives are due to some systematic disturbance. One such observed systematic disturbance was identified at the beginning of most log files. Such turbulence may occur in some signals when starting up and tearing down the environment. This is especially true for simulated environments and may generate enormous amounts of false fail indications. A situation as the one in Figure 3 with massive streaks of failed evaluations should lead to the suspicion of such problems at either system startup or shut down. However, cutting the log in either end is not without risks, especially when a passive test case relies on a sequence to occur or complete. Even trivial sequences like a button toggle may wreck the entire evaluation of a passive test case, e.g., if the first press in the log file is ignored, so this activity must be done in the context of each passive test case.

To investigate whether false positives stem from such startup problems, we need to bring up a detailed evaluation view over several logs, preferably from different testing sessions.

Figure 5 illustrates such a view for spotting systematic fails. The figure shows the evaluation of the G/A REQ-281

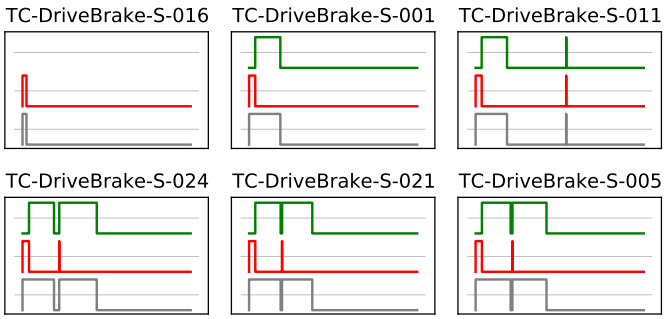

Fig. 5: Systematic Fail Analysis, REQ-281

over six logs[3]. Each sub-figure shows the evaluation over the corresponding log file. The plots in each sub-figure show the guard passes, and fails as binary signals. The gray signal at the bottom (guard interval) goes high whenever the system is in a state where the requirement should be validated. The (red) signal in the middle graphs goes high whenever the requirement is not met during the guard interval (fail). Similarly, the topmost (green) signal goes high, where the requirement *is* met (pass).

Focusing on the very first part of each log evaluation, we observe a fail at the very beginning of each evaluation for REQ-281. Further, the fail interval is too long to be explained by natural latencies or sampling effects in the system. According to our domain expert, this requirement was sensitive to some startup adjustments in the simulator. In this study, only 4 out of 17 passive test cases were susceptive to such startup disturbances, emphasizing the recommendation against a default ignore. For those four test cases, we ignored the first 32 seconds using the keyword `ignore`, as shown at line 2 in Listing 7.

```
1  // Tuning for Simulated Rig
2  ignore < 0s
3  allow 500ms fail
4  const TIMEOUT = 500ms
5  // ...
6  'REQ-456-A2' =
7  while Cab_doors_closed_and_locked_in_cab_A2 == false
       and
8        Standstill == true
9        and
10       Bypass_active_in_ready_to_run == true
11 shall
12       Traction_safe_command == false
13       and
14       bitmask(CabDrOp,
15               MWT_traction_block_reason) == CabDrOp
16       and
17       Allow_traction == false
18 within TIMEOUT
19 // ...
```

Listing 7: Example G/A With Timing Specifications

This removed all confirmed false[4]-positives during startup. However, still, many passive test cases failed while they should pass. The result of this tuning step is presented in Figure 4.

[3]We could fit six logs into the paper figure. For real, it may be beneficial to analyze more logs.

[4]All fails were confirmed to be false positives.

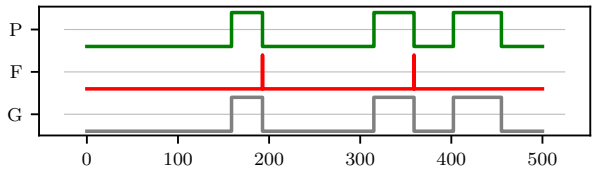

Fig. 6: Example of Natural Latency Fails of a (Should-Be) Passing G/A

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	-	-	-	P	-	-	[P]	-	-	-	-	-	-	-
REQ-246	-	-	-	-	-	-	-	[P]	-	-	-	-	P	P
REQ-248	-	-	-	-	-	-	-	-	[P]	-	-	-	-	-
REQ-253	-	[P]	-	-	-	-	-	-	-	-	-	-	-	-
REQ-254	-	-	[P]	-	-	-	-	-	-	-	-	-	-	-
REQ-255	-	-	-	[P]	-	-	-	-	-	-	-	-	-	-
REQ-258	[P]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-259	[P]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-260	-	-	-	-	[P]	-	-	-	-	-	-	-	-	-
REQ-281	P	P	P	P	P	P	-	-	-	-	-	[P]	P	-
REQ-283	-	-	-	-	-	-	-	-	-	[P]	-	-	-	-
REQ-290-A1	-	-	-	-	-	-	-	-	-	-	[P]	-	-	-
REQ-290-A2	-	-	-	-	-	-	-	-	-	-	[P]	-	-	-
REQ-349	P	P	P	P	P	[P]	P	P	P	P	P	P	P	P
REQ-350	-	-	-	[P]	-	-	-	-	-	-	-	-	-	-
REQ-456-A1	-	-	-	-	-	-	-	F	-	-	-	-	P	[F]
REQ-456-A2	-	-	-	-	-	-	-	P	-	-	-	-	-	[F]

Fig. 7: Assertion Latency (`within`) Tuned

3) Latencies and Sampling Effects on Assertions: Aside from the startup turbulence, one distinct type of false fail, observed in the study, is shown in Figure 6 at approximately 300s. In the example, the guard starts with a very short fail period but then passes until the end of the guard period[5].

A typical example in this study is REQ-245; when the doors are open, a start-inhibit signal should be set. According to our observations, when the doors are open, it takes a (short) while until the start-inhibit signal is set. Even though the requirement contains timing information, using G/As to explore assertion latencies, together with proper domain knowledge, allows establishing better, worst, and acceptable latency margins. Further, some safety-related signals are only indirectly observable at the system level, which adds to the specified timing in the requirement. The tester needs to judge whether the latencies are reasonable or not, a process that would be facilitated by such a tool. Although we call it assertion latency, other probable sources of such delays include sampling issues or different time-domain effects (e.g., A machine is used to simulate parts of the system and a real-time simulator other parts).

A tool that discovers harmless assertion latencies needs to make sure that the fail starts simultaneously as the guard and that it is followed by a substantially larger pass period to rule out other, potentially severe fails. Confirmed harmless assertion latencies can be ignored by adding the `within`, as

[5]Except for a possible guard latency as described in the next section.

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	P	P	P	P	[P]	P	P	P	P	P	[P]	P
REQ-245	-	-	-	P	-	-	[P]	-	-	-	-	-	-	-
REQ-246	-	-	-	-	-	-	-	[P]	-	-	-	-	P	P
REQ-248	-	-	-	-	-	-	-	-	[P]	-	-	-	-	-
REQ-253	-	[P]	-	-	-	-	-	-	-	-	-	-	-	-
REQ-254	-	-	[P]	-	-	-	-	-	-	-	-	-	-	-
REQ-255	-	-	-	[P]	-	-	-	-	-	-	-	-	-	-
REQ-258	[P]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-259	[P]	-	-	-	-	-	-	-	-	-	-	-	-	-
REQ-260	-	-	-	-	[P]	-	-	-	-	-	-	-	-	-
REQ-281	P	P	P	P	P	P	-	-	-	-	-	[P]	P	-
REQ-283	-	-	-	-	-	-	-	-	-	[P]	-	-	-	-
REQ-290-A1	-	-	-	-	-	-	-	-	-	-	[P]	-	-	-
REQ-290-A2	-	-	-	-	-	-	-	-	-	-	[P]	-	-	-
REQ-349	P	P	P	P	P	[P]	P	P	P	P	P	P	P	P
REQ-350	-	-	-	[P]	-	-	-	-	-	-	-	-	-	-
REQ-456-A1	-	-	-	-	-	-	-	P	-	-	-	-	P	[P]
REQ-456-A2	-	-	-	-	-	-	-	P	-	-	-	-	-	[P]

Fig. 8: Guard Latency (`allow`) Tuned

demonstrated on line 18 in Listing 7, the passive test case ignores latencies fails up to the specified time limit (500ms). Although it is tempting to use a sizeable global time limit, this may conceal severe problems in the system. On the other hand, setting the `within` too narrow will give false fails due to variations in the active test case logs. The added `within` statements (between 200-500ms) solved all assertion latencies as defined above and shown in Figure 7. REQ-254 and REQ-456 are still failing.

4) Latencies and Sampling Effects on Guards: Another distinct source of false-positives observed in the study is exemplified in Figure 6 at approximately 180s. The guard *mostly* passes, but fails for a *tiny* part at the *end* of the guard interval. It appears as if the guard condition ended too late. This behavior was observed in, e.g., REQ-245, where traction seems to be allowed just before the doors were closed and locked. Again, exploring fail intervals, together with proper domain knowledge, allows establishing an acceptable range for guard latencies. Although we call it guard-latency for simplicity, there may be other sources of such delays as for the assertion latencies. A tool that automatically finds such fails would match all fail-intervals within a guard-interval, immediately preceded by a pass-interval and ending where the guard ends.

Confirmed harmless guard-latencies can be ignored by adding the `allow`, as demonstrated at line 3 in Listing 7. The keyword specifies to the G/A to ignore guard latencies up to a specified time limit similar to `within`, although it is tempting to use a sizeable global time limit, this may conceal severe problems in the system. On the other hand, setting the `allow` too narrow will give false fails due to variations in the active-test-case logs.

In our particular case, the engineers concluded that the delays up to twice the sample time were acceptable. Given an analysis of the failing G/As, `allow` was added up to double the sampling time. The remaining test cases were left unchanged. The added slack addressed all remaining latency/sampling

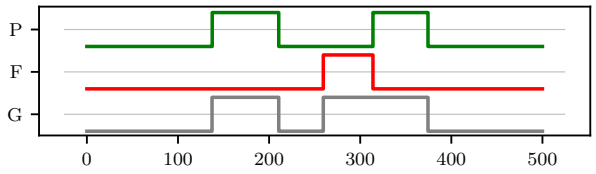

Fig. 9: Fail in test case that requires root cause analysis.

effect problems for all our G/A over the available log files, as presented in Figure 8.

5) Remaining Issues and Root Cause Analysis: When all of the above standard procedures fail to explain a fail interval in a passive test case, the chances are that we are facing a real bug, unknown feature interaction, or insufficient/faulty requirement description. In either case, the remaining fails need to be examined closely. Figure 9 shows such a fail from an early stage in the translation process. In this case, the failure turned out to be a misinterpretation of the signal redundancy, which led to the use of an AND operator instead of an OR operator.

G. Phase III - Final Evaluation

The 116 SAFE requirements were examined and are estimated to be applicable for 74 of the 116 reviewed SAFE requirements, which is in line with a previous case study using an early prototype [13].

In the remaining section, we show the results from two logged manual sessions performed by an expert. The first session is performed on a well-tested and released system in a HIL-rig, as opposed to our gold standard produced in a completely simulated environment. The expert tester's goal is to cover as much of the functionality during the session. The purpose of the first session is to validate that the false positives have been tuned away and demonstrate how passive test cases can be used to understand requirement coverage. A fail for the REQ-246 revealed that the timing was slightly different for the HIL-rig signals compared to the simulator. After adding an allowance of 250ms, the G/A passed. The results are presented in Figure 10. Each row shows the evaluation of a G/A, as translated in previous sections. A green "P" shows that the G/A could be evaluated, did not fail, and passed at least once. The "Evaluation Details" column show how many times the G/A was activated (guard activations) and the total number of fails and passes during the logged session. The gray "-" denotes a G/A that could not be evaluated. In this case, due to a missing signal. Having missing signals is a common situation since there is often a restriction on how many and which signals can be logged. Since the expert did not log that signal, the corresponding G/As could not be evaluated. However, the remaining G/As show that the achieved coverage was at least 9 out of 18 G/As. Since this system was extremely well tested, the absence of fails shows that the tuning has been successful. A closer look at REQ-244 in Figure 10 reveals that the requirement has been tested nine times during the session, and REQ-281 six times.

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

Fig. 10: Resulting G/As, Expert Session I (Well-Tested).

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

Fig. 11: Resulting G/As, Expert Session II (Fault-Injected).

In the second session, the test engineer injected two intermittent faults in the system that would be difficult to detect using traditional scripted testing. The result is presented in Figure 11. Again, a signal was not logged, so the gray "-" does not provide any information. There are, however, some yellow "-" (Guard never activated); a never activated guard means that the tester has not covered the corresponding requirement in the session, which was confirmed by the expert tester in this session. The red "F" shows where the G/A detected violations of the requirements. Analysis of the failed requirements (G/As) against the injected faults concluded that the two faults affected the system in a way that make it violate exactly these four requirements (G/As) during the short time the faults were injected.

V. Related Work

This work relies on three areas: a testing paradigm, specification, and tools supporting the use of this paradigm. In the first area, the testing paradigm, we rely on the work of independent guarded assertions [7], [8], introduced as a means of increasing the parallelism in automotive testing. This concept is similar to the passive testing approaches listed in [2] and run-time verification [18], [19]. Using these methods heavily relies on a formal specification of the test cases. Although there exist attempts to offer pre-defined patterns and even graphical representations to facilitate the formalization of either requirement *or* test cases [5], [4], they still expose the underlying formalism. The approach of the suggested T-EARS language instead strives for being simplistic and close to the requirements' text and is based on an Easy Approach to Requirements Syntax (EARS), created at Rolls-Royce to improve expressing natural language requirements [12]. The choice of EARS is motivated by its usefulness for large scale requirements in multiple domains [20] [21]. Also, in the third area, there are attempts to overcome practical issues. Related examples here include specification pattern support [22], [23] and creating monitors, similar to the guarded assertions in Matlab [24].

The approach of using guarded assertions has been previously evaluated in [8] but the guarded assertions were specified using a model-checker and based on test cases, while this paper uses T-EARS and translates directly from requirements. Another similar work by Pudlitz et al. [25] has focused on making requirements testable by using a markup language [26]. While this work relies on annotations of natural language text and the authors are able to analyze how well test cases are aligned with the requirements, T-EARS supports the temporal specification of requirements and allows a seamless integration with a passive testing tool chain (i.e., SAGA tool chain).

VI. Conclusions and Future Work

In this paper, we conclude that passive testing is adequate for testing the studied system-level requirements in a real industrial setting. In addition, we show how passive testing can be used to understand requirement coverage and finding faults. However, we observed a risk that false positives quickly affect the results without proper tuning of the passive test cases. Further, our results suggest that mapping of the logical (abstract) signals to technical (concrete) signals is a major challenge. Thus, we further improve the translation process's test case tuning steps and suggest some lessons learned when applying passive testing in a real industrial context.

The translation and tuning of requirements are done manually, which can be a time consuming activity, so future improvements include automating it, e.g., using automated latency analysis. In addition, more empirical work on the cost-effectiveness of passive testing in the embedded software industry is needed as well as further support for industrial uptake and adoption.

Acknowledgement

This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement Nos. 871319, 957212 and from the Swedish Innovation Agency (Vinnova) through the XIVT project.

REFERENCES

[1] K. M. Brzeziński, "Active-passive: On preconceptions of testing," *Journal of Telecom. & Info. Tech.*, no. 3/2011, pp. 63 – 73, 2011.

[2] A. R. Cavalli, T. Higashino, and M. Núñez, "A survey on formal active and passive testing with applications to the cloud," *Annals of telecommunications*, vol. 70, no. 3–4, pp. 85–93, 2015.

[3] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *The 21st International Conference on Software Engineering*. ACM, 1999.

[4] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar," *IEEE Transactions on Software Engineering*, vol. 41, no. 7, pp. 620–638, 2015.

[5] P. Filipovikj, M. Nyberg, and G. Rodriguez-Navas, "Reassessing the pattern-based approach for formalizing requirements in the automotive domain," in *The Intl. Requirements Eng. Conference*. IEEE, 2014.

[6] P. Mouttappa, S. Maag, and A. Cavalli, "Monitoring based on IOSTS for testing functional and security properties: Application to an automotive case study," in *The Annual Computer SW and App. Conference*, 2013.

[7] T. Gustafsson, M. Skoglund, A. Kobetski, and D. Sundmark, "Automotive system testing by independent guarded assertions," in *The Intl. Conference on SW Testing, Verification and Validation Workshops*, 2015.

[8] G. Rodriguez-Navas, A. Kobetski, D. Sundmark, and T. Gustafsson, "Offline analysis of independent guarded assertions in automotive integration testing," in *The Intl. Conf. on Embedded SW and Systems*, 2015.

[9] D. Sundmark and A. Kobetski, "Parallelization of integration tests, prestudy report," SICS, Tech. Rep., May 2013.

[10] F. Daniel, E. Eduard, A. Wasif, S. Daniel, G. Thomas, and K. Avenir, "From natural language requirements to passive test cases using guarded assertions," in *The Intl. Conf. on SW Qual., Rel. and Sec.* IEEE, 2018.

[11] D. Flemström, T. Gustafsson, and A. Kobetski, "Saga toolbox: Interactive testing of guarded assertions," in *The IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2017.

[12] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (ears)," in *The 17th IEEE International Requirements Engineering Conference*. IEEE, 2009.

[13] D. Flemström, T. Gustafsson, and A. Kobetski, "A case study of interactive development of passive tests," in *The 5th International Workshop on Requirements Engineering and Testing*. ACM, 2018.

[14] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[15] M. Tiegelkamp and K.-H. John, *IEC 61131-3: Programming industrial automation systems*. Springer Publishing Company, Incorporated, 2010.

[16] J.-L. Boulanger, *CENELEC 50128 and IEC 62279 standards*. John Wiley & Sons, 2015.

[17] M. Staron, *Action Research as Research Methodology in Software Engineering*. Cham: Springer Intl. Publishing, 2020, pp. 15–36.

[18] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Jrnl. of Logic and Alg. Prog.*, vol. 78, no. 5, pp. 293–303, 2009.

[19] K. Selyunin, T. Nguyen, E. Bartocci, and R. Grosu, "Applying runtime monitoring for automotive electronic development," in *International Conference on Runtime Verification*. Springer, 2016.

[20] A. Mavin and P. Wilkinson, "Big ears (the return of" easy approach to requirements engineering")," in *The 18th IEEE International Requirements Engineering Conference*. IEEE, 2010.

[21] A. Mavin, P. Wilksinson, S. Gregory, and E. Uusitalo, "Listens learned (8 lessons learned applying ears)," in *The 24th IEEE International Requirements Engineering Conference*, 2016.

[22] P. Filipovikj, T. Jagerfield, M. Nyberg, G. Rodriguez-Navas, and C. Seceleanu, "Integrating pattern-based formal requirements specification in an industrial tool-chain," in *The 40th IEEE Annual Computer Software and Applications Conference*. IEEE, 2016.

[23] W. Miao, X. Wang, and S. Liu, "A tool for supporting requirements formalization based on specification pattern knowledge," in *The Intl. Symp. on Theoretical Aspects of SW Engineering*. IEEE, 2015.

[24] J. Zander-Nowicka, I. Schieferdecker, and A. M. Perez, "Automotive validation functions for on-line test evaluation of hybrid real-time systems," in *The 2006 IEEE Autotestcon*, 2006.

[25] F. Pudlitz, F. Brokhausen, and A. Vogelsang, "What am i testing and where? comparing testing procedures based on lightweight requirements annotations," *Emp. SW Eng.*, vol. 25, pp. 2809–2843, 2020.

[26] F. Pudlitz, A. Vogelsang, and F. Brokhausen, "A lightweight multilevel markup language for connecting software requirements and simulations," in *The International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2019.