From Natural Language Requirements to Passive Test Cases using Guarded Assertions

Daniel Flemström*, Eduard Enoiu*, Wasif Azal*, Daniel Sundmark*, Thomas Gustafsson¶ and Avenir Kobetski||
* Mälardalen University, Sweden, Email: firstname.lastname@mdh.se
¶ Scania CV AB, Sweden, Email: thomas.gustafsson@scania.com
|| RISE SICS AB, Sweden, Email: avenir.kobetski@ri.se

Abstract—**In large-scale embedded system development, requirements are often expressed in natural language. Translating these requirements to executable test cases, while keeping the test cases and requirements aligned, is a challenging task. While such a transformation typically requires extensive domain knowledge, we show that a systematic process in combination with passive testing would facilitate the translation as well as linking the requirements to tests. Passive testing approaches observe the behavior of the system and test their correctness without interfering with the normal behavior. We use a specific approach to passive testing: guarded assertions (G/A). This paper presents a method for transforming system requirements expressed in natural language into G/As. We further present a proof of concept evaluation, performed at Bombardier Transportation Sweden AB, in which we show how the process would be used, together with practical advice of the reasoning behind the translation steps.**

I. INTRODUCTION

In today's vehicular software systems, new sensors and advanced features such as advanced driver-assistance systems are introduced at an accelerating pace. The resulting complexity has implications on the number of possible combinations and situations that need to be tested.

Test automation is often assumed to be the only reasonable approach to mitigate this development. However, there are still a number of challenges that need to be solved when automating test cases from requirements. Some of these challenges stem from the widely used industrial practice of describing system level requirements in natural language.

While there have been some efforts in trying to automatically extract information from system requirements to build test cases e.g., [1], there are, to our knowledge, no such approaches that have reached a level of maturity close to what is needed in an industrial context.

In this paper, we present a process for transforming system level requirements to passive test cases using a systematic process that consists of five main activities. The first two activities involve the analysis of requirements and the creation of abstract test cases. The next two activities produce prototype executable passive test cases, before the final step of validating and tuning these into their final form. We envision that a systematic process will make the translation from system requirements to concrete and executable passive test cases less time-consuming and error-prone. Further, this method could be used to support and maintain the traceability between requirements and test cases as well as integrated into a more general software test process improvement approach [2].

In addition to the above stated process, we present a proof-of-concept evaluation of this method, performed at Bombardier Transportation Sweden AB using an industrial subsystem responsible for the headlight control functionality.

The paper is organized as follows. Section II provides an overview of testing vehicular systems, formalizing requirements and passive testing. Section III describes the concept of guarded assertions and the SAGA tool chain. Section IV and Section V present the process of deriving and defining passive test cases using guarded assertions, as well as an evaluation of this process on an industrial system. In Section VI we briefly review the related work. Section VII discusses the results and implications of using this process in practice before we conclude the paper in Section VIII.

II. BACKGROUND

As many vehicular functions are targeted by strict standards such as ISO 26262 for automotive and EN 50126/50128/50129 (Reliability, Availability, Maintainability, and Safety (RAMS)) for trains, the vehicular systems industry must be able to argue that the system has been properly and sufficiently tested against the requirements. This puts high demands on test efficiency & effectiveness, requirements alignment and traceability between requirements and test cases [3].

Using system level requirements to derive test cases come with certain challenges: requirements and the test cases may (i) differ in level of abstraction, (ii) make use of different ontologies [4], (iii) be written by (and for) people with different competencies [5]. In addition, deriving test cases from natural language requirements suffer from well-known general issues when using natural language, such as ambiguity, vagueness, and, wordiness [6]. Any solution to these challenges would involve extracting relevant information from the requirement text, such that the tester can understand *what* to test, *how* to test, and under *which circumstances* it makes sense to test.

Transforming natural language requirements into formal representations has shown to be an effective way of *verifying* that the requirements are met, but so far it has been too difficult for the general tester and the requirements engineer to *express* requirements or test cases in a formal way [7], [8], [9].

Although there have been attempts to automatically process natural language requirements using machine learning [1],

[10], few of these attempts have survived long outside an academic setting [11] and not to the level of creating concrete and executable test cases.

Other, manual attempts such as the method proposed by Filipovikj et al. [12] aims to facilitate the translation between the natural language to temporal logic by offering pre-defined patterns, mapping common natural language constructs to temporal logic patterns. In the aforementioned work, the authors propose 17 patterns that support the translation from structured English language into formal specifications. However, using these patterns typically requires tool support to be practically useful [9], [13]. Another approach is to start from the point of the domain knowledge of the tester, trying to simplify the specification language itself so it can be used by the average tester. While many such attempts use a graphical representation of the language [11], the SAGA[1][14] approach instead offers interactive visual feedback while writing the test cases with a minimal textual specification language. The approach starts close to the domain knowledge of the test engineers with the goal of allowing the testers (and the requirements engineers) to think of the system as they are used to, without the burden of mastering temporal logic languages.

Passive testing is an approach where the 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. A more general in-depth review of passive testing methods can be found in [15]. In previous work on the SAGA approach, interviews with practitioners [16] have shown that passive testing using **guarded assertions** [14], [17], [18] seems to contribute to small tests, preferably one test per requirement, which should facilitate both alignment as well as traceability.

III. Guarded Assertions, T-EARS and the SAGA Tool Chain

The concept of guarded assertions [17] was introduced as an approach for passive testing. Initially, the target was system level testing in the vehicular domain. The concept relies on the separation of the input stimuli (that affects the system state) and the test oracle (i.e., that decides if a system requirement is fulfilled or not). In this paper we focus on the oracle, expressed as a guarded assertion (G/A). A G/A is effectively an executable test case that can be evaluated on arbitrary log files extracted from the SUT, given that the required signals for test case execution have been recorded.

A G/A consists of one guard expression G and one assertion expression A. The guard expression controls when the assertion expression A is evaluated. Depending on the type of guard expression, the resulting verdict is either a sequence of pass/fail *events*, or, a sequence of fail/pass *intervals*, where *fail* or *pass* verdicts are determined by the assertion expression A being evaluated to true (i.e., pass) or false (i.e., fail). The possible types of expressions are further discussed in Section III-A.

[1]Situation-based Integration Testing of Automotive Systems using Guarded Assertions

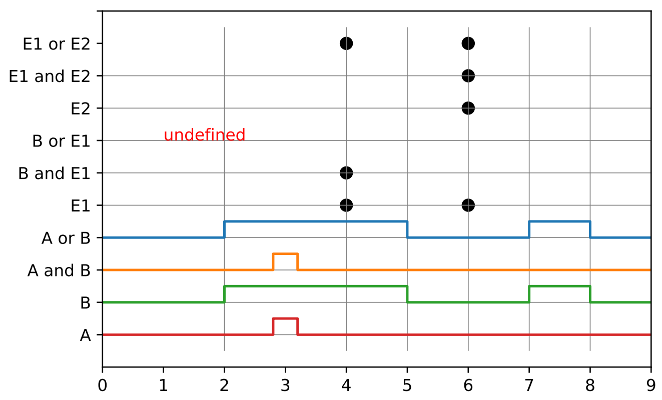

Fig. 1: Example of operations between expressions. A high signal on the y-axis illustrates the intervals when the expression evaluates to true and a dot denotes an event that only represents a point in time (x-axis).

Before examining the technical details of G/As, consider the following illustrative example: *"whenever the brake pedal is pressed, the brake light should be lit"*. Assuming that we successfully created a guard and an assertion expression for this example, the guard expression G would decide whether the brake pedal is pressed or not (a sequence of time intervals where the guard is true), and the assertion expression A would evaluate to true whenever the brake light is lit. For each guard interval, as long as A is true, the test is considered to be passed. Conversely, if A is false any time during the guard interval, the test has failed during the interval where A was false. Outside the guard intervals the result of the assertion expression is not evaluated.

The SAGA approach [14] is used to express such guarded assertions. The SAGA approach is supported by a tool chain consisting of an interactive test case editor as well as a description language (i.e., the T-EARS language). This language is based on the Easy Approach to Requirement Specification (EARS) language [6]. Below, we present the T-EARS language in detail, beginning with the principles and concepts of the expressions in Section III-A, followed by a description of the three kinds of guarded assertions that are possible to express in Section III-C through Section III-D. The three kinds of guarded assertions correspond to three of the EARS requirement patterns (i.e., event-driven, state-driven and ubiquitous). Which pattern to choose depends on the expected behavior of the test object as described by a requirement.

A. Concepts and Principles

When working with guarded assertions, we consider three expression types that can be used for the guard expression and the assertion expression. The considered types are: $EventExpression$, $IntervalExpression$, and $TimeExpression$.

An $IntervalExpression$ can be built from combining signals, values or another $IntervalExpression$ using the

T-EARS built-in operators `<`, `>`, `==`, `!=`, `and`, `or`, `xor`. The result of evaluating an $IntervalExpression$ on a test log is a sequence of time intervals where the expression evaluates to true. For simplicity such a sequence can be thought of as a digital signal (e.g., the result of $a > 3$, where a is a signal) that is true during the intervals in the sequence.

An $EventExpression$ evaluates to a sequence of *events*. In this context, each unique event maps to a single point *point* in time in the test log. An $EventExpression$ can be built from combining an $IntervalExpression$ with another $EventExpression$, using the T-EARS built-in operator `and`. The result will be a set of events confined to the intervals of the $IntervalExpression$. Using the binary signal analogy, we keep the events for which $IntervalExpression$ is true. An $EventExpression$ can also be built using the T-EARS built-in functions `rising_edge(`$IntervalExpression$`)` or `falling_edge(`$IntervalExpression$`)` that are explained further in the following examples. Figure 1 illustrates event expressions (E_1, E_2), signals (A, B) and the result of different combinations.

Lastly, the temporal specification, $TimeExpression$, in T-EARS applies to both guards and assertions and uses the following keywords:

- *`within`*: the time within which a condition must hold.
- *`for`*: minimum time that a condition must hold.

B. Event-driven Guarded Assertion

An event driven guarded assertion observes the system at discrete points in time and is described by the following pattern:
`when` $EventExpression < TimeExpression >$ `shall` $AssertionExpression < TimeExpression >$

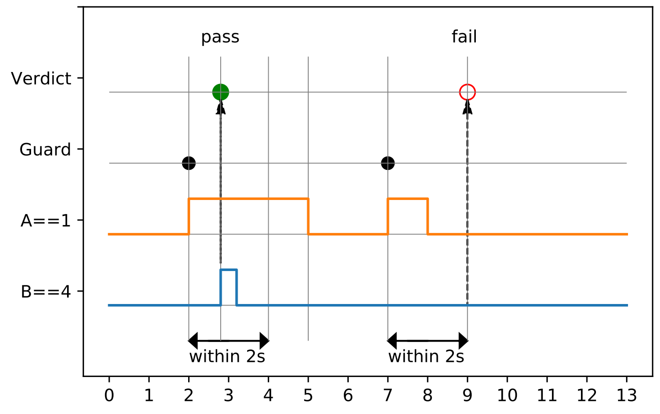

Fig. 2: Event driven G/A example: `when rising_edge(A==1) shall B == 4 within` 2s.

The guard expression $EventExpression$ evaluates to a sequence of guard events (i.e., single value at discrete time points) for which the result of the $AssertionExpression$ can be either an $IntervalExpression$ or an $EventExpression$ that should be observed. The result (verdict) is a sequence of pass or fail events. The optional $TimeExpression$ allows concentrating on the guard expression. There are currently three sources of events for the guard expression:

- **Time-out**: In some cases we need to detect when a signal has been stable during a minimum amount of time. Using the pattern "`when` $IntervalExpression$ `for` $timeout$" we can specify that an event occurs each time an interval from $IntervalExpression$ is long enough.
  If the interval is long enough, the event is elicited at the end of the timeout interval. If we choose to use the signal analogy, we get an event at the end of the timeout if and only if the "signal", represented by $IntervalExpression$, is true during the whole timeout.

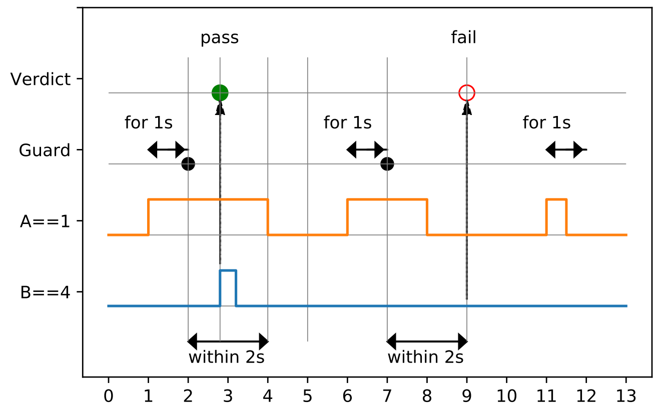

Fig. 3: Event driven G/A example with timeout: `when A==1 for 1s shall B == 4 within` 2s.

In Figure 3 the guard *`when`* $A == 1$ *`for 1s`* evaluates to one event, one second after each time step in which the clause $A == 1$ has changed from false to true, but only if $A == 1$ stays true during exactly one second. In our example, only the first two pulses (i.e., 1-4s and 6-8s) are long enough. The last pulse (i.e., 11-11.5s) is too short and does not result in any event. The verdict is thus evaluated at two points in time (i.e., 3 and 7s). At time point 2s the guard starts observing whether the expression $B == 4$ is true within the specified 2 seconds. After one second, $B == 4$ is true and the verdict is pass. In the same way, the guard starts observing the expression $B == 4$ at time point 7s, but the expression never evaluates to true, and thus, a fail verdict is reported after two seconds (as specified in `within` 2s). For a tester, the interval between the guard event and the reported (fail) verdict can be helpful to narrow down the root cause of the failure.
- **Edge Detection** (rising/falling) can be used on any binary signal expression using the T-EARS built-in function `rising_edge` and `falling_edge`.
- **Boundary Crossing**: events can be constructed by using edge detection on a signal boundary: e.g., "`when`

`rising_edge( S > 10 )`". This would evaluate to a sequence of events with one event each time the signal S crosses the boundary from $S <= 10$ to $S > 10$. A `falling_edge` would, conversely, elicit one event each time S crosses the boundary from $S > 10$ to $S <= 10$.

C. State-driven Guarded Assertion

A state driven guarded assertion observes the system while the system is in a particular state and is described by the following pattern:
`while` *IntervalExpression* < *TimeExpression* > `shall` *IntervalExpression* < *TimeExpression* >.

In practice this means that a state-driven guarded assertion takes as input a sequence of guard *intervals* and returns the continuous evaluation of the assertion expression for these intervals in time. The result is one sequence of passed intervals, and one sequence of failed intervals. It should be noted that one guard interval can result in several pass/fail intervals since the assertion expression is evaluated continuously during the whole of each guard interval.

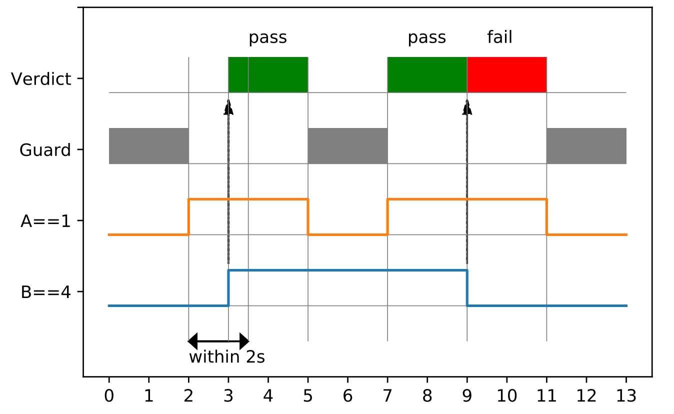

Fig. 4: State-driven G/A example: `while A==1 shall B == 4 within 2s`.

In some cases, we need to define "entering a relevant system state" separately from "leaving that state". The main reason is that describing the guard interval with one long expression would make the expression complex and hard to comprehend. For these purposes, the built in function `between(S,E)` can be used for constructing interval sequences from event sequences. The function takes two event sequences as arguments. Consider the following example: $S = \{s_1, s_2, s_3\}$ and $E = \{e_1, e_2, e_3\}$. Further, assume that s_2 occurred before e_1 and e_2 occurred before s_3. The S would then evaluate to the time intervals $\{(s_1 - e_1), (s_3 - e_3)\}$. We note that the start event s_2 occurred before e_1 and was thus ignored. In the same way, e_2 occurred before s_3 and was also ignored.

D. Guarded Assertions for Ubiquitous Requirements

This is a special case of the state driven guard where the guard expression always evaluates to true. An ubiquitous guard is thus described by the pattern *`while` 1==1*.

IV. Translating Requirements into T-EARS Guarded Assertions

In order to derive passive test cases in the form of T-EARS guarded assertions from system level requirements, we need the following: i) the set of available system requirements together with ii) the corresponding design documentation for the, iii) current implementation and, iv) a selected feature of the system under test (SUT) that we want to create test cases for. The result of the process is a set of Guarded Assertions that can be automatically evaluated on a set of log files from the SUT. Such log files may come from executing a manual or automated test scenario on, either a software simulator (also known as Software In the Loop (SIL)) or from a hardware rig (also known as Hardware In the Loop (HIL)) or from a real vehicle. The proposed process does not take the stimuli, required to put the system in a testable state, into account. Instead, we rely on a set of already available executed and recorded test scenarios, containing realistic input sequences involving the selected feature. We further assume that all the required signals have been recorded when executing the test input.

The process structures the work into the following five main activities mirrored in Figure 5: (A) *Requirement Analysis*, (B) *Abstract G/A Construction*, (C) *Implementation Analysis*, (D) *G/A Concretization* and (E) *Tuning and Validation*.

The process starts with the activity *Requirement analysis*, denoted (A) in Figure 5, where information required to understand the feature under test, and its relations to other features, is collected. The details on this activity are presented in the Section IV-A.

The collected information is formalized and eventually formed to the first version of the Guarded assertions in the sub sequent activity *Abstract G/A construction*, denoted (B) in Figure 5. In this context, abstract means that it cannot be automatically evaluated. This is because the requirements are often based on logical entities such as velocity or distance, rather than the physical signals or events to be observed. Keeping the logical names of involved entities also simplifies communication with the requirements engineers. The details on this activity are presented in the Section IV-B.

In order to evaluate the guarded assertions, each abstract entity needs to be mapped to the actual implementation by first analyzing the design documentation. This analysis work ranges from simple lookups of a given signal from a subsystem, to fusing information from different sensors and performing necessary calculations. This activity is denoted Implementation Analysis (C) in Figure 5 and is further described in Section IV-C.

The concretization activity, denoted (D) in Figure 5, includes the mapping between abstract entities and their physical counterparts (result from the implementation analysis). The result of this activity is an initial version of the guarded assertion(s), ready to be evaluated in the interactive SAGA tool. This activity is described in Section IV-D.

Writing test cases containing timing information that result in correct verdicts based on temporal specifications is a

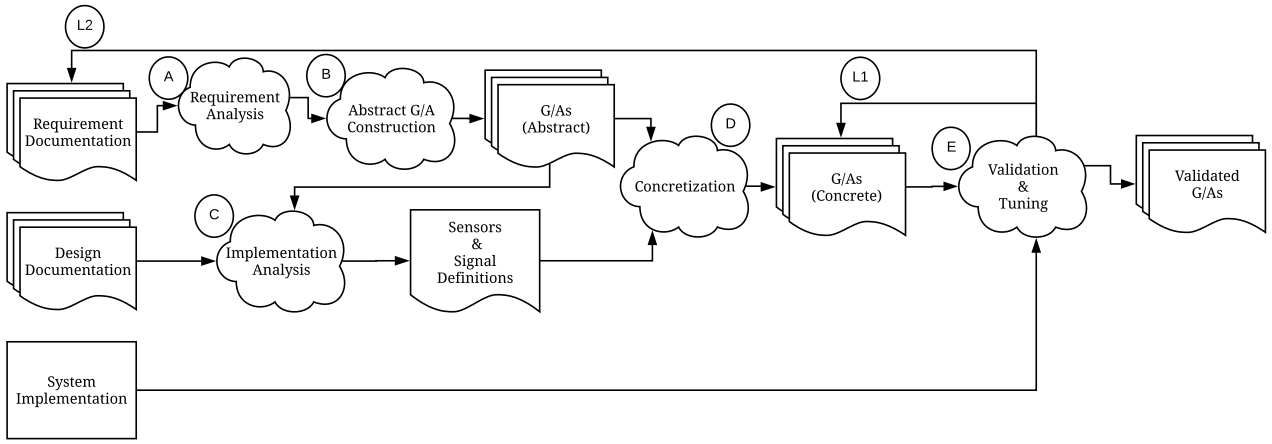

Fig. 5: Process overview— A Roadmap for Transforming System Requirements to Guarded Assertions as Passive Test Cases.

difficult and error prone task. Therefore the process includes a validation and tuning activity, denoted (E) in Figure 5, where the guarded assertions can be evaluated on different log files to check that the verdict is what is expected. From this activity, there is a feedback loop (L1 in Figure 5) to update the expressions in the guarded assertion.

Since the input stimuli used when producing the test logs typically involves simultaneous use of several features, unknown functional interference is likely to affect the test verdict. Thus, one purpose of the validation activity is to try to disclose as much as possible of unknown functional interference. The feedback loop L2 in the figure, illustrates the process of including the missing requirements for those features and updating the Guarded assertion to consider the identified functional interaction. Another reason to go back to the requirements is unfeasible temporal specifications, such as time limitations that can never be met. The Validation and tuning activity is described in Section IV-E

In the remaining sections, each step of the process will be further explained in detail.

A. Requirements Analysis

The basic approach for this step is to establish all related requirements and their dependencies as well as identifying the guard and assertions expressions using the identified logical signals and values.

Information gathering takes as input the feature that is tested and the all system requirements and its related documentation including all available information related to know dependencies and relations between the SUT and other functions and environment entities. Each selected feature aims to accomplish a set of required goals and is provided explicitly in the requirement to a test engineer. The output of this step is a subset of the system requirements and related documents that sufficiently describes the feature under test and how it influences the other features. In this study we consider two distinct attributes in the requirement documentation: the features and the dependencies between features. In turn, feature dependencies are functionalities required for requirement analysis, test execution, and are part of the guarded assertions.

The problem with identifying feature dependencies is that these need to be distinguished and taken into consideration before a test engineer constructs a G/A. One important difference between traditional testing of checking the observable behavior based on a sequence of input stimuli (i.e., in the form *stimuli-check-stimuli-check sequence*) and passive testing using G/As, is that G/As are completely independent of the input stimuli sequence and cannot actively affect the state of the system. Instead, a realistic input sequences is assumed to be constructed outside the scope of writing the G/As. Thus, it is likely that other features may interact with the verified property at any point in time during system execution.

Consider the following “brake light” example: *A request for brake light should be sent when the brake pedal is pressed more than 10%*. In a traditional representation of a test case, this requirement could contain enough information for writing a standalone test case, since the test sequence can be designed to limit the stimuli to the system to “pushing the brake pedal”, thus guaranteeing that no other system function can interfere. However, when using G/As, we need to find other system features affecting the feature under test. In the “brake light” example this could be an automatic emergency brake feature that is also requesting the brake light. As a consequence, relevant requirements related to this feature needs to be included as well. It should be noted that, in practice, one of the major issues in developing large embedded systems is that feature dependencies are not easily accessible for a test engineer due to the sheer size of the system and its complexity. Passive testing has been suggested as way to facilitate the identification of such dependencies, since these test cases will often fail due to feature interactions during system execution. These feature interactions can be either the result of an unknown and albeit correct interdependency indicating a missing requirement or a fault in the actual software.

After this gathering of information, the chosen requirements are extracted from the requirement documentation and interdependences are identified, we guarantee that only the sub-set of requirements related to the feature under test are assessed.

B. Abstract G/A Construction

Using the set of requirements from the previous step, we analyze these requirements with respect to the test objects (*what*) of the asserted properties, their expected behavior (*how*), and, the circumstances (*when*) under which the behavior is expected. The (*when*) part results in one or more guard expressions, while the (*what*) and (*how*) parts result in one or more assertion expressions. The construction of the G/A is performed sequentially, taking each requirement one by one, by accessing only the information available. At this level, the guard and assertion expressions are built using *abstract* signals and abstract values so they can easily be traced back to the original requirement. Thus, the G/As are abstract since these cannot be directly evaluated using passive testing by assessing it using log-file extracted from the SUT. These G/As are using abstract instead of concrete elements, because the abstraction level and the focus of the G/A is directly linked with the selected features under test. Throughout this paper, the term 'abstract G/A' will be used to denote both the deliberate omission of details in the G/A construction and the encapsulation of details by means of high-level signal and values representing states and events. Details that are not encoded at this stage obviously cannot be included on the grounds of the requirement documentation. In addition, it entails the use of bridging the different levels of abstraction between the requirement document and the the SUT: the G/As need to be concretized before it is usable on the SUT. The hope is that an engineer can split the inherent complexity of a system into an abstract G/A first, and at a later stage analyze the implementation and driver components needed to perform G/A concretizations. On the other hand, the abstract G/A must be sufficiently precise to serve as a basis for passive testing. This means that these should be complete enough in terms of asserted and expected behaviors as well as temporal constraints and rules.

In practice, an engineer performing the abstract G/A construction based on the requirement analysis step needs to fulfill the following four activities:

1) **Language Harmonization.** In some occasions, the level of ambiguity of a requirement make it difficult to extract the needed information. Language harmonization is used to discover the meaning of a particular requirement.
2) **Extraction of G/A Information.** This activity describes the process of gathering the test objects, behaviors and constraints needed to formalize a set of requirements using the following steps: (i) *Extraction of Asserted Properties and Test Objects,* (ii) *Extraction of Expected Behaviors,* (iii) *Extraction of Conditions and Rules* and (iv) *Extraction of Temporal Constraints.*
3) **Pattern Selection.** This activity deals with the selection of a guard pattern and an assertion expression type needed for determining the actual formalization.
4) **Abstract G/A Formalization.** In this activity a requirement is formalizable if there is a guard pattern and assertion type that captures its semantics. The guarded assertion is formalized by defining the abstract signals and signal feature shapes and finally composing the T-EARS expressions for the guard and the assertion. The timing-specification of these signals is also taken into account. Practically, the following steps have to be fulfilled: (i) *Definition of Abstract Signals and Shapes,* (ii) *Guard Expression Composition,* (iii) *Assertion Expression Composition,* and (iv) *Temporal Specification.*

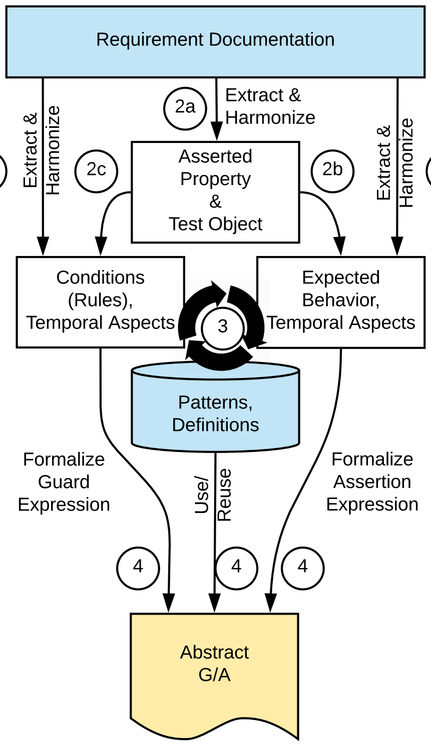
Requirement Documentation
2a
Extract & Harmonize
2c
Extract & Harmonize
2c
Asserted Property & Test Object
2b
Extract & Harmonize
2b
Conditions (Rules), Temporal Aspects
3
Expected Behavior, Temporal Aspects
Patterns, Definitions
Formalize Guard Expression
Use/ Reuse
Formalize Assertion Expression
4
4
4
Abstract G/A

Fig. 6: An Overview of the Requirement Analysis and Abstract G/A Construction.

In the rest of the section we present how the different activities of constructing an abstract G/A are performed. First, a set of related requirements together with other sources of information (e.g, corresponding manual or automated test cases, sequence diagrams) and the set of already defined G/As and G/A expressions are used as an input to the abstract G/A construction. The result of this step is a set of abstract G/As corresponding to the features under test.

The Figure 6 outlines the requirement analysis process. The arrows describe the information flow and the numbers on the arrows denote the listed activities resulting in the entities shown as boxes (i.e., test objects, conditions, expected behaviors and finally the abstract G/As). When selecting the pattern for the guard and assertion expressions, the available patterns, conditions as well as the expected behavior must be considered together. This activity is illustrated in Figure 6 with the iteration symbol (3).

In more detail, the analysis starts with extracting the asserted properties, (2a) in Figure 6. The asserted properties describe what is tested (test object), how the test object should behave, and, when it should behave in that way. Within the asserted property, the test object captures the system-related entity, addressed by a certain requirement (e.g., "The **doors** shall be locked within 3s when vehicle is underway"). However, when using natural language the requirement is analyzed to recognize the test object. In practice, this can be identified in a sentence as a noun. Although there are tools developed to to automatically detect these nouns [1], *which* of the nouns in the sentence to choose is heavily depended on the context and requires human reasoning and domain knowledge. One way is to identify what is the test goal and what is actually tested for a particular requirement. By finding this goal in the requirement we identify the test object.

Once the test objects are determined, the corresponding expected behaviors are extracted and identified (i.e., (2b) in Figure 6) by capturing the expected system behavior for each test object (e.g., "The doors shall **be locked** *within 3s* when vehicle is underway"). The context indicates whether the behavior is discrete by denoting a discrete event (e.g., a signal going from low to high can be described as single value at a discrete time step) or continuous requiring a continuous state or function over time (e.g., a signal having a specific value for a certain interval). In this context, this behavior representation is important when selecting the T-EARS pattern for the final formalization of the assertion expression, (3) in Figure 6. For example, phrasal verbs, such as "be locked" typically indicate a *state type* assertion expression, while verbs directly representing actions, such as "lock" or "begin" may indicate the use of an *event type* assertion expression[2].

The requirement example "The doors shall be locked within 3s when vehicle is underway" contains a temporal constraint (i.e., "within 3s") which is annotated together with the expected behavior and the selection of the assertion type (i.e., state or event).

Before formalizing the expected behavior to an assertion expression, the conditions under which the behavior are established by directly extracting from the requirement written in natural language, (2c) in Figure 6. In our example, the condition is described at the end of the sentence representing the requirement example: e.g., "The doors shall be locked within 3s *when vehicle is underway*". Once again, the behavioral context is required to decide if the condition describes a discrete event or a continuous state. In this example "is underway" indicates that a certain property is true over an interval and can thus be considered as an abstract continuous system *state*, as opposed to "starts moving" which indicates a discrete momentary *event* (e.g, events only require single values at discrete time steps).

While performing the extraction activities (i.e., (2a)-(2c) in Figure 6), language harmonization of the G/A information makes sure that all chosen entities from the requirements text align with some commonly agreed taxonomy or pattern. This activity reduces the risk of defining duplicate abstract signals, guards or assertions and facilitates reuse of formalized expressions.

When all necessary information is extracted from the requirement text, harmonized and analyzed, it is used for choosing appropriate guard and assertion expression patterns, (3) in Figure 6. Depending on the available information on the expected behavior, we choose to use an event or a state assertion expression pattern. This decision in turn influences which of the guard patterns that are applicable. Since this decision depends on the available information in the requirement, several iterations may be required until a satisfactory result is achieved.As a general rule, an event triggered guard can only result in a verdict that is a series of fail and pass verdicts at discrete points in time. In practice, this means that we can assert that a system event leads to another event or signal change. Any other behavior that happens outside this point in time is not checked. On the other hand, a state-based guard results in a continuous verdict over a whole continuous range of time steps while the system is in the described state. This means that any glitches or irregularities in the test object will result in a partial fail verdict. Finally, the formalization and pattern selection step should result in the creation of an abstract guarded assertion, (4) in Figure 6.

C. Implementation Analysis

Abstract guarded assertions are mapped to the actual implementation by analyzing the design documentation. Design documents contain information about the implementation elements of a system and their interactions, as well as the relation between signals and requirements. An engineer needs to take this information into account and identify the given signals and their characteristics. By identifying suitable implementation and concretization details from these design documents, we define a set of sensors and signals related to the feature under test that is needed to fully describe the entities used in the abstract G/A (e.g., `vehicle_underway`, or `train_between_platforms`).

For example, an abstract signal `vehicle_underway` can be defined in multiple ways, and the design documentation may provide a set of possible sensors (e.g., `SpeedSensor1`, `StandStill`) represented using physical identities, relevant values and scaling property for each source of information. In this step alternative concrete data channels for the abstract signals, events and their values are collected by identifying relevant sensors, signals or other system state information sources that will be used in G/A concretization.

D. G/A Concretization

The concrete G/A is built upon actual signals and events containing sensors and signals identified in the previous activity. The result is a set of guarded assertions that can be evaluated on a system log using the following steps: (i) *Identification of data sources* and (ii) *Value interpretation*.

The input of this concretization step is an abstract G/A definition. For each information entity in the produced abstract

[2]Described $IntervalExpression$(state) and $EventExpression$(event)

G/A, domain knowledge is used to identify the underlying meaning of each signal. As an example, an abstract signal `vehicle_underway` can be read from different speed sensors on the wheels in the vehicle or from a higher level system output on a vehicle control unit. In this case, we can use the highest level possible, given that all steps between the source sensor on the different wheels are validated through a series of G/As, and therefore ensuring the signal propagation. However, ultimately, this decision depends on the overall context of the requirement and requires deep domain knowledge. The next step is to interpret abstract values such as "high" or "low" in the context of the selected source of information. This depends on the overall context and the measured unit and scaling properties of the selected data source. Oftentimes, the same abstract entities are used over and over again, so the definitions of the abstract signals/events should be accumulated in a global list and thoroughly documented (i.e., in which context and testing level these are relevant).

E. Tuning and Validation

This last activity takes as input the executable G/A. The G/A is validated using the following two phases: (i) *Interactive Validation* and (ii) *Temporal Tuning*. The goal of the interactive validation is to ensure that the expressions of the guard and the assertion work as expected. Any missing signals or requirements, will show as unexpected fails. The temporal tuning allows experimentation to ensure e.g., that timeouts are adequately dimensioned.

In practice this means that the list of concrete signals is logged during a manual exploratory test session. Using the interactive SAGA tool, the G/A can be applied to a log file and the guards can be examined to observe that the asserted time periods behave as expected. Any adjustments are immediately reflected in the tool-output. In the same way, the assertions can also be tuned in the editor and adjusted to ensure that the requirement is fully covered by the set of G/As produced.

V. Proof of Concept Evaluation

In the following example we present the analysis of an authentic, albeit slightly simplified set of requirements for the selected feature "headlight" of a metro train at Bombardier Transportation Sweden AB's part of a Train Control Management System (TCMS). TCMS is the name of the software-based system in charge of the overall train control including the operation-critical, safety-related functionality. TCMS is a system with multiple types of software and hardware components that controls all parts of train operation, including propulsion, line voltage and passenger comfort systems. The software part of TCMS is part of a distributed system running on multiple hardware units distributed across the whole train.

The "headlight" functionality works as follows: At each end of a metro train, there is a cab where the driver controls the train, but for clarity, we consider only one of the cabs. The full beam of the headlights is controlled using a pushbutton on the driver's desk in the cab. One push to turn it on, and the next to turn it off. However, the full beam is only active whenever the train is ready to run. To get the train in that state, the train driver needs to perform a specific sequence of actions (e.g., log in to the train panel). The train is normally taken out of that state when the driver deactivates the cab, or by other system functions, such as emergency brake or safety features, overriding the manual control.

The proposed process described in Section IV is applied on this "headlight" example by focusing on each step of transforming system requirement to guarded assertions and running these as passive test cases on the actual system implementation.

A. Requirements Analysis

Following the process outlined in Section IV-A, we analyze the system requirements and extract all the ones that relate to the "headlight" function. For the sake of simplicity, we only consider the feature "full beam" of the lights. The set of identified requirements consist of two textual requirements presented in Table I. The two requirements define when the full beam of the train headlight should be on and off respectively.

B. Abstract G/A Construction

As described in Section IV-B, the goal of this activity is to systematically identify, organize and harmonize information to finally create one or more abstract guarded assertion(s).

1) Language Harmonization: Since the requirements in our example are written in natural language, we needed to understand the underlying meaning of the text as well as identify entities of interest. We performed this activity in parallel with the extraction activities: expected behaviors, conditions and temporal constraints for the behaviors and conditions.

2) Extraction of G/A Information: The goal of this activity is to extract and structure the information in order to facilitate the upcoming formalization step, where the abstract guarded assertion is formed.

Extracting the asserted properties and test object: Given an overall read through of the requirements, we identified the asserted property for REQ-1 as "full beam of the headlights being on", under the conditions described on lines 1.a through 1.c in Table I . The asserted property in REQ-2 was the inverse case, describing when "full beam of the headlights being off" given the conditions on lines 2.a through 2.c in the same Table. This helped us to identify the test object, "full beam lights", which answers the question *what* should be tested?

Extracting Expected Behaviors: For the identified test object "full beam lights" we extract the expected behaviors "start to request" and "stop to request" on line 1 and 5 respectively, and add this to Table II in the column "expected behaviors". These are our raw material for the assertion expressions and answers the question *how*. There is no information on timing requirements for these behavior. Thus, we note N/A in the Time column.

Extracting Conditions and Rules: For each listed behavior in Table II, we identify the circumstances under which these behaviors are expected. The sentence "when all of the following is fulfilled" in row 1 of Table I guide us to add

TABLE I: Outcome of the Requirements Analysis: The set of collected requirements for the train head lights.

1	REQ-1	TCMS shall start to request full beam lights when all of the following is fulfilled:	
2		1.a	full beam control on driver's desk is pressed (rising edge-triggered)
3		1.b	the cab is Ready to run
4		1.c	no request for full beam request is currently active.
5	REQ-2	TCMS shall stop to request full beam lights when all of the following is fulfilled:	
6		2.a	full beam control on driver's desk is pressed (rising edge-triggered)
7		2.b	the cab is Ready to run
8		2.c	request for full beam request is currently active.

TABLE II: Final Result of Extracted Conditions and Expected Behaviors for REQ-1&2, test object "full beam light"

Row	Expected Behavior ⟶ Assertion	Time	Conditions(rules) ⟶ Guard Expression	Time
1			full beam control on driver's desk is pressed (rising edge-triggered) **and**	
2	start to request full beam lights	N/A	the cab is Ready to run **and**	N/A
3			no request for full beam request is currently active.	
4			full beam control on driver's desk is pressed (rising edge-triggered) **and**	
5	stop to request full beam lights	N/A	the cab is Ready to run **and**	N/A
6			request for full beam request is currently active.	

the contents of the rows 2-4 to the conditions for "start to request full beam" to Table II. We further add *and* between the individual rows since the requirement says "all of them" should be fulfilled. These rules are our raw material for the guard expressions. There is no timing requirements for the conditions either. We repeat the procedure for REQ-2 as well, and add the results to the same table.

3) Pattern Selection: By consulting other testers with necessary domain knowledge, we interpret the expected behavior "start to request", as the full beam lights going from "not requesting" to "requesting full beams". This is something that happens momentarily. The rules of T-EARS says that we in such cases need to create an event driven guarded assertion.

4) Abstract G/A Formalization: Before we can formalize the guard and assertion expressions in Table II we have to harmonize the language and define abstract signals we can use in the expressions for the guards and for the assertions.

Definition of Abstract Signals and Shapes: Reading Table II we notice that "full beam request" (lines 3 and 6) has the same meaning as "full beam lights" (in the expected behavior column). Our domain knowledge guides us to the interpretation that it is an abstract signal, telling that the full beam should be lit. We note both wordings in the same cell in Table III and assign an abstract signal name, **`full_beam`**, to facilitate the upcoming formalization process. This cannot be automatically decided, but comes from domain knowledge and asking around among other experienced testers. In the same way, we cluster each entity surrounded by a solid box in Table II (if they have the same meaning) and finally assign an abstract signal name to each cluster. Correspondingly, the behaviors (marked with a dotted box in Table II) were harmonized and assigned abstract values such as **`on`** or **`off`**.

Guard Expression Composition: In Section V-B3 we concluded that we need an event triggered guard expression. The T-EARS pattern for an event triggered guard is `when`

TABLE III: Harmonized Language and logical definitions, requirement REQ-1 and REQ-2.

Entity ⟶ Abstract signal	Behaviors ⟶ Abstract Values
- full beam lights - full beam request **`full_beam`**	- start to request - request for... active **`on`**
	- no request for - stop to request - request for ... removed **`off`**
- full beam control on drivers's desk **`fb_btn_pressed`**	- is pressed (rising edge-triggered) **`rising_edge(fb_btn_pressed)`**
- the cab **`cab_ready_to_run`**	- Ready to run **`true`**
	- Ready to run ... is removed **`false`**

$EventExpression$. The task is to translate the rows 1-3 of Table II to an event expression assigning abstract guard clauses to the rows :

G1 `rising_edge(fb_btn_pressed)` - (events)
G2 `cab_ready_to_run == true` - (binary signal)
G3 `full_beam == off` - (binary signal)

Revisiting the requirement again reveals that all three clauses should hold. Given the T-EARS documentation in Section III, the first expression (G1) results in an event sequence. One event each time the button is pressed. However the button pushed events should only be considered whenever the two other clauses (G2 and G3) holds. Therefore, we need to discard all the events where (G2 and G3) evaluates to false. The overloaded **and** operation in T-EARS does this for us. Thus, the event triggered guard clause results in the definition `E_full_beam_on` in Listing 1 row 2. The corresponding steps were undertaken for the REQ-2, resulting in the define `E_full_beam_OFF` on row 12 in the same listing.

Assertion Expression Composition: The goal of this step is to form abstract assertions from the expected behaviors listed in Table II. We use the abstract signal and value names from the Table III for the requirements, resulting in the expression on row 9 in Listing 1 for REQ-1 and row 19 for REQ-2.

Temporal Specification: In our requirements, there are no temporal requirements available. However, we still add "`within 0.2s`" as an illustration of a temporal specification in Listing 1 (line 9 and 19). The practical meaning is that, for each event of the guard expression, where we allow a delay of 0.2 seconds for the assertion clause to evaluate to true.

```
# REQ-01
def E_full_beam_ON
   rising_edge(fb_btn_pressed == on) and
   (cab_ready_to_run == true and
   full_beam == off)

when E_full_beam_ON
shall
   full_beam == on within 0.2 s

# REQ-02
def E_full_beam_OFF
   rising_edge(fb_btn_pressed == on) and
   (cab_ready_to_run == true and
   full_beam == on)

when E_full_beam_OFF
shall
   full_beam == off within 0.2 s
```

Listing 1: Resulting Abstract G/A, first iteration

Improving the Solution: Although the resulting guards in Listing 1 cover the requirements REQ-01 and REQ-02, a drop in the observed signal would go unnoticed.

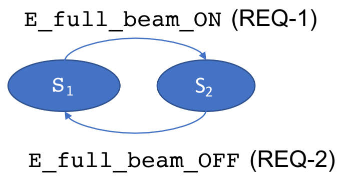

Fig. 7: Considering State Transitions Instead of Events.

In the G/A definitions in Listing 1 , we concentrated on the state transitions, rather then within which intervals the system is expected to remain in one state or another. If we consider full beam being "on" or "off" as abstract system states, as in Figure 7, we can use the built-in function `between`, to construct such periods in time. Reinterpreting REQ-1 and REQ-2 together now gives that full beam should be on (represented by the state S_1) from the point in time where the `E_full_beam_ON` event is encountered, until the `E_full_beam_OFF` event is encountered:

`between(E_full_beam_ON, E_full_beam_OFF)`

We also need to consider when to assert the state S_2, representing where the full beam should be off. Inverting the expression, would make sure that no other function by accident turns on the lights when they should be off:

`between(E_full_beam_OFF, E_full_beam_ON)`

```
# REQ-01, REQ-02
while between(E_full_beam_ON, E_full_beam_OFF)
shall
   full_beam == on within 0.2 s

while between(E_full_beam_OFF, E_full_beam_ON)
shall
   full_beam == off within 0.2 s
```

Listing 2: Abstract G/A, Second Iteration

C. Implementation Analysis

The goal of this step is to list all abstract signals and values, and find out which subsystems, sensors, vehicle control units that can provide the necessary information and also which concrete signal name to read. We also need to understand and interpret the signals in relation to the specification. In our case the information comes from different subsystems, which is reflected in the naming schemes. In our case, a corresponding physical signal can be found in the current implementation of the SUT for all of our abstract signals. We also need to make sure that the interpretation of the signal values match the interpretation used in the abstract G/A. By consulting the current implementation documentation, we create the following map [3]:

- `full_beam = Head_light_full_beam_on`
- `cab_ready_to_run = TC_BI_CCUS_S_RdyToRn`
- `fbt_btn_pressed = DriversDesk.a1.heljuspb`
- `on = 1`
- `off = 0`
- `true = 1`
- `false = 0`

D. G/A Concretization

The goal of the concretization step is to create 'defines' for the abstract signals that we defined in the implementation analysis. Given the names of the physical signals found, we can add the following definitions (Listing 3) before the abstract G/A definition to make it a concrete guarded assertion.

```
def full_beam
   Head_light_full_beam_on
def cab_ready_to_run
   TC_BI_CCUS_S_RdyToRn
def fb_btn_pressed
   DriversDesk.a1.heljuspb
def on
   1
def off
   0
```

Listing 3: Concretization Mapping

E. Tuning and Validation

In order to validate any timing specifications, and to make sure that the G/A is complete, we record the identified signals using an existing manual test case, where we turn on and off the lights while performing a realistic driving cycle.

The recorded log can then be loaded into the tool, where we also enter our abstract GA together with the defines.

Now the tool can be used for adjusting or adding missing timing information or signals that needs to be considered. Since there are inevitable delays in the system we must allow for a slight delay in turning on the lights. When evaluating

[3]The Signal names have been shortened and where necessary obfuscated.

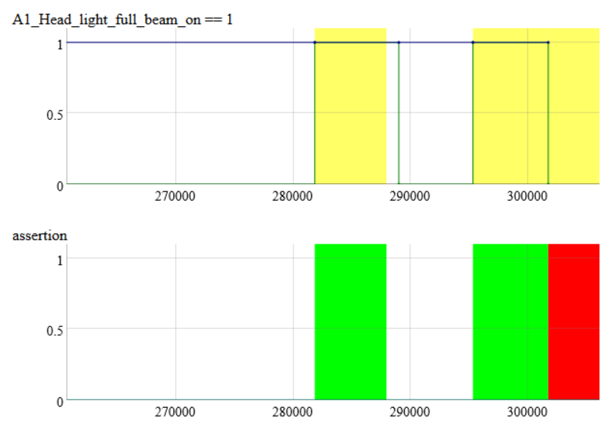

Fig. 8: Cut-out snapshot of the SAGA tool. Demonstrating how a missing requirement may cause the G/A to report failure. Yellow denotes active guard interval. Green/red indicates passed/failed assertion.

the G/A against the test log, we discover that the lights go off without anyone pushing the light button (Figure 8). This happened when the emergency stop was tested, with the effect that ready to run was removed for the cab. The feedback loop L2 in Figure 5 reveals a third (missed) requirement. Whenever the cab is not ready to run, the lights should be off. Adding this information to the OFF event, the Guarded Assertion reported all success for the given log file, without having to redo the manual test case and the recording.

VI. Related work

In automotive system testing, scenario-based testing is commonly used, where a scenario is typically a sequence of system events [19]. This typically results in hardcoded scripted test cases that are difficult to test for non-determinism [18] as well as for interactions among various configurations and signal combinations. The proposed independent guarded assertions [20] allow for parallel execution of test cases, with frequent evaluation of assertions that are not typically captured in requirements and thus allows for testing of functional interactions. The concept of guarded assertions is similar to the use of *Automotive Validation Function* by Zander-Nowicka et al. [21] where an *Automotive Validation Function* makes assertions if pre-conditions are true. The difference between an automotive validation function and an independent guarded assertion is that the later can be used to concurrently test several functions. The concept of declarative testing [22] is also related where, for example, test automation is modularized into *Answer*, *Executor* and *Verifier* entities. The *Answer* represents the goal of a scenario, *Executor* transforms *Answer* declaration into executable operations while *Verifier* compares the final actual state with the original *Answer* entity. Lastly, as we mentioned in Section III, the concept of passive testing also has interesting parallels with the concept of independent guarded assertions.

There exists some previous work on the translation of natural language requirements in to representations suitable for model-based testing. For example, Silva et al. [23] show the translation process from natural language text to colored petri nets and Sarmiento et al. [24] show a transformation of requirements into UML activity diagrams. Recently, some research has focused on using natural language processing techniques to generate test cases from functional requirements [25] [26].

The EARS (Easy Approach to Requirements Syntax) language, created at Rolls-Royce to improve expressing natural language requirements [6], is also central to the translation process explained in this paper. There is some evidence on the usefulness of EARS for large scale requirements from multiple domains [27] [28]. Similar to our effort, different other extensions to EARS have been proposed, such as EARS-CTRL by Lúcio et al. [29] for writing and analyzing EARS requirements for controllers and Adv-EARS for derivation of use case models by Dipankar et al. [30].

VII. Discussion

The SAGA approach of using guarded assertions to define passive test cases is an attempt to bridge the gap between traditional scenario-based testing (which, in our experience, is relatively straightforward and widely used in the vehicular industry, but limited in terms of test variability, coverage of functional interference and test efficiency), and more formal approaches (that may be exhaustive, rigorous and more robustly defined, while at the same time seldom seem to reach a broad usage among testing and development practitioners).

A. *Towards Industrial Adoption of SAGA and G/As*

In order for a new testing method to reach industrial use, it is not sufficient that it is *useful* beyond today's best practices. It also needs to be *usable* by the people that are intended to use it. The focus of this paper is the description of the process of translating natural language requirements into passive test cases in the form of guarded assertions. Along with a mature toolchain and underlying method, we consider the existence of such a structured guiding process to be a key element in the industrial update and adoption of passive testing.

In accordance with most contemporary approaches to development of complex systems, the proposed process supports the iterative nature of complex problem-solving. Given the inherent temporal logic nature of many vehicular system requirements (e.g., whenever A happens, B should occur within C ms), combined with the inherent complexity of temporal logic, we believe that an interactive and incremental way of expressing passive tests is likely to be more applicable in an industrial context.

B. *Limitations*

While we are continuously improving this situation by working with several test engineers at multiple industrial partners, our experience of translating requirements to guarded assertions (either by doing it ourselves or by observing the results of it being done by testers at our industry partners [16]) is still limited. Similarly to each guarded assertion derived through the proposed process, it is likely that the process itself will need to be refined and made more efficient and effective.

VIII. Conclusion and Future Work

In this paper, we have presented and demonstrated a detailed process for translating natural language requirements for vehicular systems to passive test cases, expressed in the form of SAGA guarded assertions. This systematic process has been established with the primary objective to support industrial uptake and adoption of the SAGA approach, and of passive testing approaches in general. Our future work will be conducted in three distinct, but interrelated, directions: (1) Further development of the SAGA methodology and toolset (including continued work on the T-EARS language and support for automated input sequence genaration), (2) empirical work on the cost-effectiveness of passive testing in the embedded software industry, and (3) further support for industrial uptake and adoption.

Acknowledgment

This work was supported by The Knowledge Foundation (KKS) through the projects 20160139 (TESTMINE) and 20130258 (Volvo Chair), by VINNOVA within the FFI program (SAGA) and from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No. 737494 (MegaM@Rt2).

References

[1] H. M. Sneed, "Requirement-based testing-extracting logical test cases from requirement documents," in *International Conference on Software Quality*, pp. 60–79, Springer, 2018.

[2] W. Afzal, S. Alone, K. Glocksien, and R. Torkar, "Software test process improvement approaches: A systematic literature review and an industrial case study," *Journal of Systems and Software*, vol. 111, pp. 1 – 33, 2016.

[3] E. Bjarnason, P. Runeson, M. Borg, M. Unterkalmsteiner, E. Engström, B. Regnell, G. Sabaliauskaite, A. Loconsole, T. Gorschek, and R. Feldt, "Challenges and practices in aligning requirements with verification and validation: a case study of six companies," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1809–1855, 2014.

[4] H. M. Sneed, "Bridging the concept to implementation gap in software system testing," in *Quality Software, 2008. QSIC'08. The Eighth International Conference on*, pp. 67–73, IEEE, 2008.

[5] M. Paulweber, "Validation of highly automated safe and secure systems," in *Automated Driving*, pp. 437–450, Springer, 2017.

[6] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (ears)," in *17th IEEE International Requirements Engineering Conference, RE'09*, pp. 317–322, IEEE, 2009.

[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, (New York, NY, USA), pp. 411–420, ACM, 1999.

[8] 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, pp. 620–638, jul 2015.

[9] P. Filipovikj, T. Jagerfield, M. Nyberg, G. Rodriguez-Navas, and C. Seceleanu, "Integrating pattern-based formal requirements specification in an industrial tool-chain," in *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, vol. 2, pp. 167–173, IEEE, 2016.

[10] S. Tahvili, E. F. M. Ahlberg, W. Afzal, M. Saadatmand, M. Bohlin, and M. Sarabi, "Functional dependency detection for integration test cases," in *Companion of the 18th IEEE International Conference on Software Quality, Reliability, and Security.*, July 2018.

[11] C. Pang, A. Pakonen, I. Buzhinsky, and V. Vyatkin, "A study on user-friendly formal specification languages for requirements formalization," in *Industrial Informatics (INDIN), 2016 IEEE 14th International Conference on*, pp. 676–682, IEEE, 2016.

[12] P. Filipovikj, M. Nyberg, and G. Rodriguez-Navas, "Reassessing the pattern-based approach for formalizing requirements in the automotive domain," in *Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE'14)*, 2014.

[13] W. Miao, X. Wang, and S. Liu, "A tool for supporting requirements formalization based on specification pattern knowledge," in *Theoretical Aspects of Software Engineering (TASE), 2015 International Symposium on*, pp. 127–130, IEEE, 2015.

[14] D. Flemström, T. Gustafsson, and A. Kobetski, "Saga toolbox: Interactive testing of guarded assertions," in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pp. 516–523, IEEE, 2017.

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

[16] D. Flemström, T. Gustafsson, and A. Kobetski, "A case study of interactive development of passive tests," in *5th International Workshop on Requirements Engineering and Testing (RET'18), June 2, 2018, Gothenburg, Sweden*, IEEE/ACM, 2018.

[17] T. Gustafsson, M. Skoglund, A. Kobetski, and D. Sundmark, "Automotive system testing by independent guarded assertions," in *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'15)*, pp. 1–7, 2015.

[18] G. Rodriguez-Navas, A. Kobetski, D. Sundmark, and T. Gustafsson, "Offline analysis of independent guarded assertions in automotive integration testing," in *The 12th IEEE International Conference on Embedded Software and Systems (ICESS)*, 2015.

[19] ISO/IEC, "Iso/iec/ieee 29119-1: 2013 software and systems engineering-software testing-part 1: Concepts and definitions," 2013.

[20] T. Gustafsson, M. Skoglund, A. Kobetski, and D. Sundmark, "Automotive system testing by independent guarded assertions," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015.

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

[22] E. Triou, Z. Abbas, and S. Kothapalle, "Declarative testing: A paradigm for testing software applications," in *2009 Sixth International Conference on Information Technology: New Generations*, 2009.

[23] B. C. F. Silva, G. Carvalho, and A. Sampaio, "Test case generation from natural language requirements using cpn simulation," in *Formal Methods: Foundations and Applications* (M. Cornélio and B. Roscoe, eds.), (Cham), pp. 178–193, Springer International Publishing, 2016.

[24] E. Sarmiento, J. C. S. d. P. Leite, and E. Almentero, "C amp;l: Generating model based test cases from natural language requirements descriptions," in *2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET)*, 2014.

[25] A. Ansari, M. B. Shagufta, A. S. Fatima, and S. Tehreem, "Constructing test cases using natural language processing," in *2017 Third International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB)*, 2017.

[26] R. P. Verma and M. R. Beg, "Generation of test cases from software requirements using natural language processing," in *2013 6th International Conference on Emerging Trends in Engineering and Technology*, 2013.

[27] A. Mavin and P. Wilkinson, "Big ears (the return of "easy approach to requirements engineering")," in *2010 18th IEEE International Requirements Engineering Conference*, pp. 277–282, Sept 2010.

[28] A. Mavin, P. Wilksinson, S. Gregory, and E. Uusitalo, "Listens learned (8 lessons learned applying ears)," in *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pp. 276–282, Sept 2016.

[29] L. Lúcio, S. Rahman, C.-H. Cheng, and A. Mavin, "Just formal enough? automated analysis of ears requirements," in *NASA Formal Methods* (C. Barrett, M. Davies, and T. Kahsai, eds.), (Cham), pp. 427–434, Springer International Publishing, 2017.

[30] D. Majumdar, S. Sengupta, A. Kanjilal, and S. Bhattacharya, "Adv-ears: A formal requirements syntax for derivation of use case models," in *Advances in Computing and Information Technology* (D. C. Wyld, M. Wozniak, N. Chaki, N. Meghanathan, and D. Nagamalai, eds.), Springer Berlin Heidelberg, 2011.