Specification of Passive Test Cases using an Improved T-EARS Language

Daniel Flemström[1 [0000−0001−8096−3592]], Wasif Afzal[1 [0000−0003−0611−2655]], and Eduard Paul Enoiu[1 [0000−−0003−2416−4205]]

Mälardalen University, Västerås, Sweden
{daniel.flemstrom}{wasif.afzal}{eduard.paul.enoiu}@mdh.se

Abstract. Test cases that only observe the system under test can improve parallelism and detection of faults occurring due to unanticipated feature interactions. Traditionally, such *passive* test cases have been challenging to express, partly due to the use of complex mathematical notations. The T-EARS (Timed Easy Approach to Requirements Syntax) language prototype was introduced to respond to this and has received positive feedback from practitioners. However, the prototype suffered from few deficiencies, such as allowing non-intuitive combinations of expressions and usage of temporal specifiers that quickly got difficult to understand. This paper builds on the T-EARS prototype and input from experienced testers on a previous iteration of the language. The collected experience was applied to a new prototype using a structured update process, including a set of system-level requirements from a vehicular software system. The results include a new, improved grammar for the T-EARS language and a description of the evaluation semantics.

1 Introduction

We trust vehicular software to be functional, safe and reliable on a daily basis. Traditionally, a great number of software tests ensure that the software works as specified. Intuitively, the more the tests can be run in parallel, the shorter each testing cycle can be and more thorough the testing. One approach that has shown promising results in dealing with this problem is passive testing using guarded assertions (G/As) [12,22]. As in contemporary passive testing or monitoring, the idea is to treat the input stimuli (that affects the system state) and the test oracle (that decides if a system requirement is fulfilled or not) independently. Consequently, if all necessary signals have been logged, passive testing allows parallel evaluation and even off-line evaluation of G/As.

A weakness with most passive testing or monitoring approaches [3] is that they rely on formal descriptions of test cases that tend to meet quite some resistance from practitioners [5,2,7] for being too complex. Although there exist predefined patterns and even graphical representations [7,2] to facilitate the formalization of either requirements or test cases, the problem of readability and traceability remains. As a reaction to such difficulties, T-EARS (Timed Easy Approach to Requirements Syntax) was proposed as an engineer-friendly approach

to writing passive test cases [8]. The T-EARS language allows writing easy-to-write and easy-to-read (executable) requirements and test cases for signal-based systems such as vehicular systems. The initial versions of the approach and the language were appreciated by the test engineers [9], but still suffered from having an experimental (very open) grammar and automatic conversions that did not always evaluate as the tester expected. Unexpected evaluation results were especially common for the timing-related keywords.

The work in this paper aims to improve the T-EARS language, prototyped in [8] and [4], so the language becomes more intuitive from a testing perspective. Primarily, these refinements concern the grammar and semantics of the language. Other refinements include suggesting a set of boiler plates to decrease the distance between the EARS patterns and the final corresponding passive test cases. Finally, the intuition and usage of the temporal specification is addressed.

The results of these refinements have been implemented and evaluated in [11]. The industrial validation part of that paper analyzed 116 safety-related requirements from an ongoing industrial project at Alstom Transport AB. The refined T-EARS language and the supporting tool-chain were found to be applicable for 64% of the studied requirements. Furthermore, an expert from Alstom Transport AB performed two testing sessions to validate the applicability of the refined T-EARS language in terms of requirements coverage and fault detection respectively. The result from the first testing session showed that the translation to T-EARS was stable for a number of requirements whereas some requirements could not be evaluated due to certain signals not being logged, which is a common situation in testing at Alstom Transport AB. In the second testing session, the expert injected faults in the SUT, known to be hard to find with traditional testing. The G/As were able to detect all injected faults. In summary, the evaluation showed how passive testing with an improved language can be used to understand requirements coverage and finding faults.

Whereas [11] focus on the overall approach and evaluation, this paper focus on providing more detailed insights into the language and how we improved it. The main contributions are (i) an Ohm grammar for the improved T-EARS language and (ii) semantics descriptions for the improved T-EARS language.

2 Background

2.1 Passive Testing

Passive testing is an approach where the test cases only observes the system under test (SUT). When a testable state is detected, further observations are done to see whether the tested requirement is fulfilled or not. The concept has been used in many variants in various domains [3]. Notably, most of these works target non-vehicular software testing, such as protocol testing in web and telecom applications and are based on formal specifications.

2.2 Guarded Assertions

The concept of an independent guarded assertion (IGA) [12] or simply a guarded assertion (G/A) was introduced as an approach for system-level, passive testing of vehicular software. A G/A is defined by a *guard* expression, G, that decides whether the *assertion* expression, A, is expected to be fulfilled or not.

Let's consider the following illustrative vehicular requirement: "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 had failed when A was false. Outside the guard intervals, the result of the assertion expression is not evaluated.

A previous attempt to express such G/As can be found in the SAGA (**S**ituation-based Integration Testing of **A**utomotive Systems using **G**uarded **A**ssertions) approach [8]. The SAGA approach is the prototype of a tool chain consisting of an interactive test case editor and a description language i.e., the T-EARS language mentioned in the next section.

2.3 Easy Approach to Requirements Syntax (EARS)

The purpose of the Easy Approach to Requirements Syntax (EARS) [16] is to provide minimal syntax, helping the requirements engineer to write natural language requirements that are less ambiguous, better structured, and less complex.

While already successful for specifying requirements [14,15], we argue that by evolving the syntax to be machine-interpretable, the quality of requirements would increase, as well as the gap between requirements and testing would reduce [17,24]. The T-EARS language [8,4,10] with the accompanied SAGA-Toolkit is a first step towards such an extension of EARS.

2.4 The Ohm Grammar Language

Ohm[1] and the Arc Ohm parser library are used for specifying the grammar and semantics of a domain specific language. Such a language is defined by i) a set of terminals, ii) a grammar and iii) a set of semantic rules. Firstly, a set of terminals (such as keywords or numbers) defines what you can write, and a set of rules define how you are allowed to combine the terminals into the different constructs of the language.

While the grammar describes all acceptable strings for the described language, it does not say anything about what it means. The interpretation (or actual meaning) of the rules is called the semantics of the language.

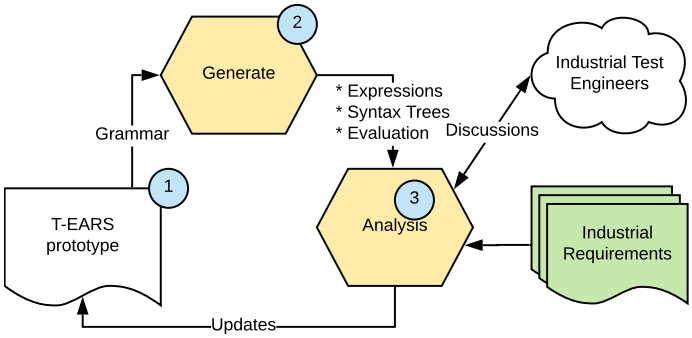


Fig. 1: Method overview

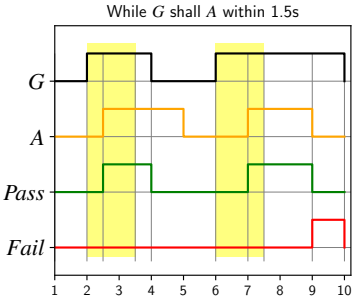
3 Method

The work of the new T-EARS version started with the prototype in [8] and a case study on the previous prototype [9] as input. The refinements were performed during a number of iterations as illustrated in Fig. 1. Each iteration started with (an updated version of) the T-EARS prototype. For the first iterations, we systematically generated possible expressions and syntax trees by hand. For each of those expressions, we (manually and independently) created a sketch of the intuitive evaluation of the expression (according to our understanding and discussions with test engineers at Scania and Alstom Transport AB). Possible and required expressions were sorted into useful expressions and expressions that should be forbidden (bad expressions). Further, to ensure the expressiveness of he language, a set of 40 safety-critical requirements from Alstom Transport AB and a complex requirement from Scania CV AB were used for the static evaluation of the language updates. The set of useful expressions was then analyzed against a set of evaluation questions concerning, e.g., usefulness, completeness and intuitiveness. Based on this analysis, the prototype grammar, the translated requirements, and the useful expressions were updated until the expressions and the grammar was consistent. This process was repeated until the requirements could be expressed as passive test cases using the updated language, leading to test cases that were easy to understand and interpret. During parallel work with industrial adoption of passive testing using the refined T-EARS language [11], a set of tuning keywords were added to ignore false fails. With all refinements in place, 116 safety critical requirements were analyzed in [11] to determine the applicability of the final results.

4 Result: The Updated T-EARS Language

T-EARS provides six boilerplates as shown in Listing 1.1. Just as EARS, T-EARS reasons about system states and system events. A system state can be

[1] https://github.com/harc/ohm



(a) BP-2:Asserting States

Fig. 2: State Boilerplate Example. Shadow= grace period of 1.5s

represented as a binary signal that is true when the system is in the specified state and false when the system is considered to be in another state. State is internally represented as a series of time intervals (while state is true), and events are represented as a series of time-stamps. In the text, we use the binary signal metaphor and intervals interchangeably.

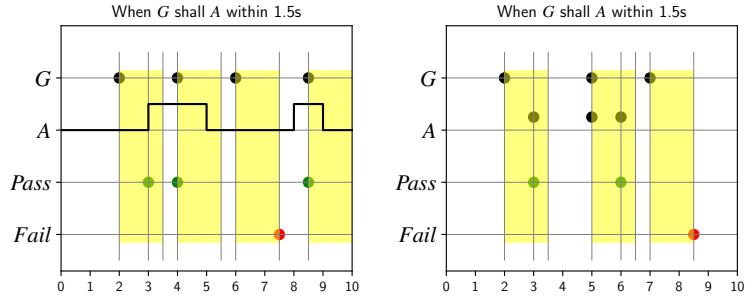
```
'Bp-1' = while true              shall <sys response state A>
'Bp-2' = while <sys state G > shall <sys response state A> within t
'Bp-3' = when <events G> shall <sys response state A> within t
'Bp-4' = when <events G> shall <response events A> within t
'Bp-5' = when <events G> shall <sys response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

Listing 1.1: Resulting T-EARS Boilerplates, sys = system

The rest of this section outlines how each EARS pattern (one through six) is realized in T-EARS. The observant reader will note that, while T-EARS, in general, follows the EARS structure and usage of keywords, the syntax `the` $<$ *system name* $>$ is not used in T-EARS. Instead, T-EARS assumes the system name to be implicit by the signal expressions to facilitate automatic evaluation of the final passive test cases. Further, when describing the patterns, the system state and response are represented as intervals or states only. More details on how to combine signals and operators to express such states and events using logged data are covered separately in Sections 4.8-4.11.

Ubiquitous: A ubiquitous requirement describes a property of the system that should always hold, e.g., "the big red emergency lamp should never be lit". In T-EARS this is realized by the first boiler plate. The result is a pass whenever the state assertion A is true and failed for not(A).

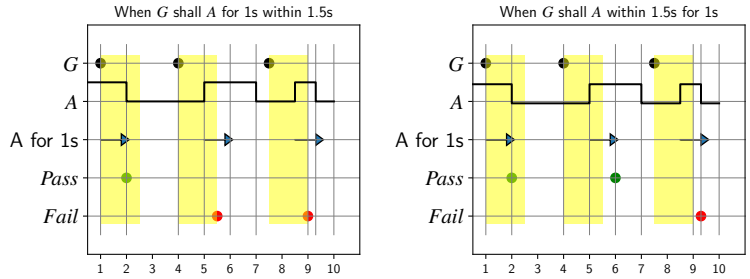
State-Driven: A state-driven requirement describes a property of the system that should hold as long as it is in a particular state. E.g., "**while** the vehicle is moving **shall** doors be locked". In T-EARS, BP-2 is used for such requirements. Figure 2 shows that, in general, during the specified guard intervals (G==true in Figure 2), a pass (P==true in Figure 2) is reported whenever the assertion is true, and a fail(F==true in Figure 2) whenever the assertion is false. While



(a) BP-3: Asserting State Response (b) BP-4: Asserting Event Response

Fig. 3: Event Boilerplates Evaluation Examples

passes are duly reported during the whole guard intervals, fails during each within-period (yellow shadow in Figure 2) are ignored. The within period starts at each guard interval and have the length specified after the `within` keyword. Outside the guard intervals, the value of the assertion is ignored.



(a) BP-5: Completed Response (b) BP-6: Started Response

Fig. 4: Boilerplate Evaluation Examples

Event-Driven: An event driven requirement describes an expected response to a (series of) discrete event(s). E.g, "**when** horn button is pushed **shall** the horn honk". T-EARS provides a few variations for this pattern for asserting event responses or a state response. The first, **BP-3** is used for asserting a system state response within a timeout whenever an event occurs. The intuition is that for each guard event, one pass is reported as soon as the assertion is true. However, if the assertion is not true any time before the timeout t, a fail is reported at g + t. Figure 3a shows how this boilerplate is evaluated for four guard events. The yellow shadow shows the within t period occurring after each

guard event. For the first guard event (at 2s) A becomes true just before the within interval ends and a pass is reported. When second guard event occurs (at 4s), A is already true and a pass is reported immediately. For the third guard event (g_3 at 6s) , the system response A does not occur within t and a fail is reported at $g_3 + t$. Finally, the last guard (at 8.5s) is immediately pass since A is already true. **BP-4** is used for asserting a system event within a timeout t as shown in Figure 3b. The semantics follows BP-3. A pass is reported if a response event (A) occurs before the within period ends, and a fail is reported at $g+t$ if no event occurred. For the first guard (2s) an event is found before the within period(in yellow) ends. A pass is reported at that time. For the second guard event (at 5s), there is an event a at 5s. Since this event occurred at the same time as the guard event, it cannot be a response to that guard event and is thus ignored. Instead next event in A that occurs at 6s yields a pass since it is still inside the within period. For the third guard event (at 7s) no response in A occurs and as for BP-3, a fail is reported at $g + t$ (8.5s). There are also two boiler plates for asserting a system state of a particular length. The first, **BP-5**, requires the system state to be tf long and finish within tw. Figure 4a shows three examples on how this boilerplate is evaluated. For the first guard event, A is already true. The for 1s period is counted from guard and results in an event if A stays true for 1.5s. This event is evaluated as in BP-3. For the next guard, A becomes true at 5s, so we start counting the 1.5s from here. However, the within period ends before the 1.5s could be completed. A fail is thus reported at $g + t$ as for BP-3. For the last guard event, A is not true long enough, but since the within period ends first, this does not matter and a fail is reported at $g + t$. BP-6 is used for asserting that a response state of (min) length tg is initiated within tw from each guard event. Figure 4b illustrates the difference between BP-6 and BP-5. With G and A the same, we note that for the second guard event, we allow the for period (the arrows in the figure) to stretch outside the within period (yellow shadow in the figure). As a consequence, the second guard events results in a pass when A has been true for 1.5s after the guard event. For the last guard event, we still get a fail, but the fail is reported because the for period could not be fulfilled (slightly later than the BP-5 example).

Option: Some requirements are only applicable to certain configurations of the SUT. E.g., "**where** the vehicle has a horn, [*horn requirement*]". In T-EARS this is accomplished by using the `where` < *boolean expression* > before a G/A boilerplate. In contrast to a guard expression (that varies over the time covered in the log-file), this is a single Boolean value that concerns the whole log-file.

Unwanted Behavior: Some behaviors are unwanted but still require a response. E.g., "**if** oil pressure is critical **then** the motor should shut down". In T-EARS, there is no `if`, or `then`-keyword, however unwanted behavior can modeled by using the existing boilerplates.

Complex: More complex requirements can be constructed by *combining* the EARS patterns. E.g **when** the honk button is pressed **while** engine is running shall horn honk. In the new version of T-EARS, nesting while and when expressions were removed in favor to stricter rules on how to combine states and

events to form guard expressions. Instead of nesting the when and while expression, above expression is expressed using the more structured rules of Events and Intervals, as *when honk button is pressed and engine is running shall horn honk*.

In the upcoming sections, we present the grammar developed to realize these boilerplates.

4.1 Keyword Terminals

The terminals grammar block defines a rule for each keyword and also groups the keywords into a logical group:

```
keyword =
/   1.      2.     3.       4.     5.      */
  where   | and | for     | true  | const
| when    | or  | within  | false | alias
| while         | longer  | inf   | def
| shall         | shorter         | events
                | than            | intervals
                | at
/*6*/
|allow|fail|ignore
```

Listing 1.2: Non-trivial Terminals

The first group of keywords outlines the G/A (e.g., `where`, `when`). The conjunctions group (`and`, `or`) allows composing expressions. The third group consists of the timing modifiers (`for`, `within` etc.). The fourth group has a set of built-in constants (`true`, `false`, `inf`). The fifth group concerns structuring the expressions (e.g., `def`, `alias`, `const`).

4.2 Structural Elements

The structural elements block defines the following main rules:

```
Constant =
    const identifier "=" (Timeout | Num | Boolean)
IntervalsDef =
    def intervals identifier "=" Intervals

EventsDef    =
    def events    identifier "=" Events

Alias = alias identifier "=" identifier
```

Listing 1.3: Structural Elements

The purpose of the rule *Constant* in Listing 1.3 is to define named constants, e.g., limits or timeouts. The constant is checked by the corresponding semantic operation of the rule where the constant is used. A constant can only be defined once within a test case context. The purpose of the rules *IntervalsDef* and *EventsDef* is to structure sub-expressions into named expressions to increase readability. A def expression can only be defined once using the same name. Further, the keywords `events, intervals` facilitates type checking while typing the expressions in the interactive editor. The expression is evaluated where

used (not where it is defined). The `alias` keyword renames an identifier. An alias can be redefined, allowing the same alias in two G/As to have different meanings. Since an alias is resolved where it is evaluated, using an alias inside a named expression offers a primitive way of user-defined functions. Another purpose of the alias keyword is to create abstractions for, e.g., release or variant of a system without changing the test logic.

4.3 Basic Data Types

There are four basic types, `Boolean, Float, Integer`, and `Time`. Listing 1.4 shows how they are defined. The example shows the Boolean type. One sub rule defines explicit usage (e.g., true,false) and one rule allows using an identifier (e.g., $--constOrAlias$). The identifier rule allows using a constant or alias (an alias is a renamed constant). There is also a main rule for how identifiers can be specified. The *identifier–quoted* allows strings in quotes that would otherwise be forbidden.

```
1  Boolean = (true | false)              --bool
2          | identifier                  --constOrAlias
3
4  identifier ="'" idstring_quoted "'"  --quoted
5             | idstring
6  //-----------------------------------------------
7  idstring = ~digit ~keyword letter+ (specialChar | alnum)*
8  idstring_quoted = (specialChar | mustQuoteChar | alnum)+
9  specialChar    = ("_" | "/" | "[" | "]" | "." | "|" | ":")
10 mustQuoteChar  = "-" | "+" | " " | "(" | ")"
11
12 sign = ("+" | "-")
13 TimeUnit = ("s"~"h / "ms"~"h")
```

Listing 1.4: Basic Data types

Examples of such strings are strings that contain spaces or keywords. The support rules below row 7 in Listing 1.4 shows the details of, e.g., sign and string handling. The tilde operator followed by the letter h (at line 13 in the listing) prevents the time unit to be confused with the keyword `shall`.

4.4 Signals Data Type

In a signal based system, the input and output consists of a set of (single value) signals that vary over time. Example signals are actuator readings, signals from other subsystems, and even sampled continuous values such as speed or temperature. These signals can be recorded into log files and fetched by name when building T-EARS expressions. Using the T-EARS editor[8,11], it is also possible to manually construct *abstract* signals to allow executable examples for higher-level requirements. In T-EARS, there are also several ways to manipulate signals as described by the *Signals* grammar block:

```
1  Signal =
2      Signal SigOP Signal        --sigOpSig
3    | SignalFunction             --func
4    | (true | false | NUM)       --constant
```

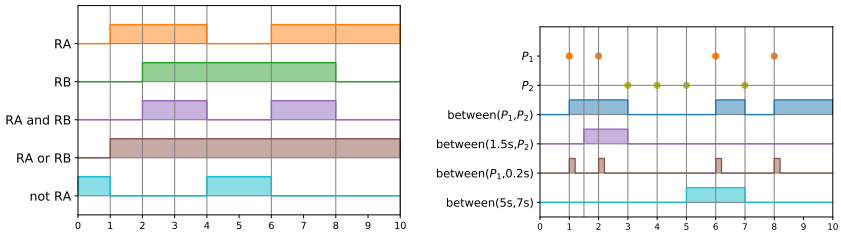
```
5    | identifier                    --sigAliasConst
6    | "(" Signal ")"               --parentheses
7
8  SigOp =
9    ("+" | "-" | "/" | "*")
10
11 SignalFunction =
12    derivative "(" (Timeout ",")? Signal ")"
13   | abs "(" Signal ")"
14   | bitmask "(" IntegerOrConst "," Signal ")"
15   | count "(" Events "," Intervals ")"
16   | maxVal "(" NonemptyListOf< Signal, ","> ")"
17   | select "(" Signal "," Signal "," Signal ")"
18   | exists "("  identifier ")"
```

Listing 1.5: The Signal Datatype

The rules Listing 1.5 defines the Signal data type. Besides the main rule *Signal*, there are two support rules: *SigOp* defines trivial mathematical operations on two signals, and, *SignalFunction*, that defines all built-in functions that return a value of type Signal.

In more detail, the Signal data type is represented by a series of samples [time, value] pairs and is denoted with the letter S (Signal). In the examples below, we use the notation $S = [s_0, \ldots, s_m]$ for a signal with $m + 1$ samples, where each sample ([time,value] pair) is noted as $s_i = [t_i, v_i]$. Logs may be sampled with a variable sampling rate, so the value between one sample is considered to be constant until the next sample.

When evaluating mathematical expressions (line 2, $--sigOpSig$), the signals are projected on a common timeline. The operator (e.g., plus or minus) is then applied on each sample along the common timeline. The $--func$ sub-rule at line 3 allows more advanced signal processing in the functions listed by the support rule *SignalFunctions*. The currently provided functions are `derivative`, a forward approximating derivative with an optional threshold to smooth out the result over several samples as $s_n(i) = \frac{v_{n+i} - v_n}{t_{n+i} - t_n}$. The threshold t makes sure to increase i from 1 until $i : t < t_{n+i} - t_n$. Increasing the threshold widens the delta in the approximation. The `abs` function processes each sample of a signal as $v_n = abs(v_n)$. The `bitmask` function returns $v_n = v_n \wedge bitmask$, applied on each sample of a signal. The `count` function takes two arguments: one Events and one Intervals argument. The result is a signal with the number of events during the interval of r_i as value. The value is constant during each interval of r_i. The `maxVal` function takes a list of Signals and returns a new Signal with the largest sample value at each sampled point in time. The `select` function selects samples from the second or third signal argument depending on the first signal argument's value. Where the first signal argument equals true, the sample from the second signal is used. Otherwise, the sample from the third signal argument is used. If the first expression is constant, only the used signal needs to be defined. The `exists` function returns a signal that is constant true if there exists a signal with the name of the given identifier. Typically, the last two functions, `select` and `exists`, are often used together to enable default values for optional signals.



(a) Intervals Conjunctions (b) The `between` Function

Fig. 5: Interval Operations and Creation. R denotes Intervals, P denotes events

The sub-rule at line four in the listing (−−*constant*) defines a pseudo-signal with a constant value. The signal is defined over the logged min and max time. It is possible to specify a binary or a numerical value.

If an identifier is specified (−−*sigAliasConst*, sub-rule at line five), this may refer to a signal name to fetch from a loaded log file, an alias, or a named constant. If the identifier is an alias, the alias is resolved until a signal name or a constant is found. A constant (sub rule −−*constant*) is interpreted as a signal with a constant value over the entire log file.

4.5 Intervals Data Type

The syntactical rules for Intervals are presented in Listing 1.6.

```
1  Intervals =
2    IntervalsExpr TimeFilter*
3    ((and|or)
4    IntervalsExpr TimeFilter* )*   --conj
5
6  TimeFilter  =
7    longer than Timeout            --atLeast
8  | shorter than Timeout           --atMost
9
10 IntervalsExpr(Interval Expression) =
11   | "(" Intervals ")"            --parentheses
12   | IntervalFunction             --func
13   | Signal RelOp Signal          --relop
14   | (true | false)               --boolean
15   | definedIntervals             --definition
16   |"[" ListOf<Interval, ","> "]" --list
17
18 Interval = "[" Timeout "," Timeout "]"
19
20 RelOp =   ("==" | "!=" | "~="  | ">=" | ">" | "<=" | "<")
21
22 IntervalFunction =
23     not "(" Intervals ")"
24   | between "(" (Events|Timeout) ","
25                 (Events|Timeout) ")"
```

Listing 1.6: The Intervals Datatype

The first rule at lines 1–4 in Listing 1.6, together with the support rule (*TimeFilter* at lines 6–9), allows filtering intervals shorter or longer than a specified threshold. The filters can be defined in any order. The rule at lines 1–4

also defines the two possible conjunctions (`and`,`or`) between intervals, shown in Figure 5a. The intuition is the same as and/or between binary signals (high inside an interval and low outside). Intervals can also be constructed by the two built-in support functions (line 12: $--func$, and lines 22–25: *IntervalFunction*). Currently, there are two such functions defined. The function `not` returns the two-complement of an interval series. The function `between` can be used for constructing intervals from Events or from one event and a constant, to create fixed length events, as illustrated in Fig. 5b.

The $--relop$ rule at line 13 together with the rule *RelOp* at line 20 defines how signals and relational operators are combined to form Intervals. The signals are projected onto a common timeline (each unique sample time from both signals) and each sample is compared using the operator. Again, it should be noted that values are not interpolated between samples. The rule at line 16 ($--list$) and line 18 (*Interval*) defines manual specification of an intervals series. The time can be specified numerically, but also by using named constants.

4.6 Events Data Type

The Events data type describes how to compose a series of system events. The rules for the Events data type are presented in Listing 1.7.

```
1  Events =
2     Intervals ForExpression          --intervalFor
3    | Events   and  Intervals         --andIntervals
4    | Intervals and Events            --intervalAnd
5    | Events or  Events               --eventsOr
6    | Events ("+"|"-") Timeout        --nudge
7    | EventFunctions                  --function
8    | definedEvents                   --definition
9    |"[" ListOf<Timeout, ","> "]"     --list
10   | "(" Events ")"                  --parenthesis
11
12 EventFunction =
13    risingEdge  "(" Intervals ")"
14   | fallingEdge "(" Intervals ")"
15   | cycle"(" (Events ",")? Timeout ")"
```

Listing 1.7: The Events Datatype

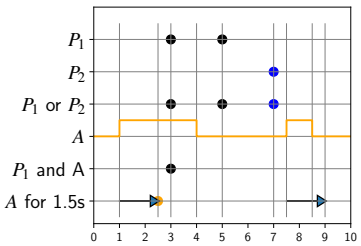


Fig. 6: Evaluation Examples of the Event Rules

The first rule at line 2 in Listing 1.7 (−−*intervalFor*) defines events as a response to a timeout on an interval. The result is one event for each interval, long enough to reach the timeout as $P = [r_s + t] \forall r : r_s + t < r_e$, where R is a series of intervals with each interval starting at r_s and ending at r_e. The next two rules at lines 3–4 (−−*andIntervals*,−−*intervalAnd*) defines the `and` operator between Intervals and Events. The intuition is that the events occurring during an interval are kept. Note that the rule P_1 *and* $P2$ is removed from the language. The reason is that events are represented by high resolution timestamps and would need to be identical to yield any results, which is not realistic. A workaround is to replace P with acceptable intervals around each event in P, as in the following example:

$$W = between(P - t, P + t) \tag{1}$$

$$P \textbf{ and } Q \Rightarrow Q \textbf{ and } W \tag{2}$$

The interval W represents an interval that is reasonably close to P. The points in Q that reside in any of these intervals will be kept. It should be noted that if the interval is constructed using the points of Q instead, the result would be the points in P that match an interval. Also, the time t needs to be sufficiently small to not create overlapping intervals in W.

Line 5 in Listing 1.7 (−−*eventsOr*) shows the only conjunction between Events. The expression P_1 *or* P_2 would evaluate to all events in P_1 and P_2, sorted and with duplicates removed.

Line 6 (−−*nudge*) shows how each event can be pushed forward or backward in time. The expression $P_1 + t$ would evaluate in a series events $[p + t]\ \forall p \in P_1$.

For more complex Events operations, there are some built-in functions that takes other data types as input and returns Events (Line 8 (−−*function*) and lines 12–15 (*EventFunction*). Currently, there are three such functions defined. The first two concerns edge detection. Detecting the edges of signals or intervals is common in creating events based on signal or interval features. Since intervals are conceptually treated as a binary signal, rising edge and falling edge correspond to each interval's start and end. The last function is requested by test engineers to ensure that cyclic events are correctly sent on the CAN bus. The first argument is an optional event series. The cycle will start at the first event in this series and continue as long as the last logged sample (in the currently loaded log). The second argument defines the cycle width.

The rule −−*list* on line 9 in Listing 1.7, allows to hand-craft en Event series by assigning *individual* time points to an Events. These time points may be either specified as (milli) seconds or by using a constant or alias for a constant.

Finally, the −−*definition* rule allows for using a defined events-expression. When evaluated, any level of aliases are resolved and eventually, the defined Events expression is evaluated.

4.7 Boolean Expressions

The EARS pattern **Option** is realized by the `where` keyword and a Boolean expression. This Boolean expression decides if the G/A should be evaluated at all or not. The grammar for Boolean is shown in Listing 1.8.

```
1  BoolExpr = BoolExpr (and|or) BoolExpr          --conj
2    | BoolExpr ("==" | "!=") BoolExpr            --eq
3    | Num RelationalOperator Num                 --op
4    | BooleanFunction                            --func
5    | Boolean                                    --boolean
6    | identifier                                 --constOrAlias
7    | "(" BoolExpr ")"                           --para
8
9  Boolean = (true | false )
10 BooleanFunction =
11     exists "(" identifier ")"                  --exists
```

Listing 1.8: The BoolExpr Datatype

4.8 Guarded Assertion Rules

There are two types of guarded assertions: The State G/A, observes the system state and expects some requirements to be held during this time. These are specified using the `while` keyword as shown in line 4 in Listing 1.9. The second type is the Event G/A that reacts to events and checks a requirement in response to the events. Event G/As are specified using the `When` keyword as shown in line 5 in Listing 1.9. Both rules are build up by a guard rule and an optional assertion rule.

```
1  GA = ((identifier "=")? Config? GuardedAssertion)
2  Config = where BoolExpr
3  GuardedAssertion =
4    while Intervals (shall IntervalAssertion)?
5  | when  Events    (shall EventsAssertion)?
6
7  IntervalAssertion = Intervals (within Timeout)?
8
9  EventsAssertion =
10     Intervals for Timeout within Timeout
11   | Events within Timeout
12   | Intervals (within Timeout)? (for Timeout)?
```

Listing 1.9: Guarded Assertions

The Events assertion is a bit more complicated with three rules. The first rule at line 10 in Listing 1.9 shows the case when the response to an event is that the system enters the state (described by Intervals) for a time (first Timeout expression). If the system is kept in the asserted state for that time, a pass is emitted for the guard at that time($g + t$). If the system does not enter the asserted state, a fail is emitted as soon as the asserted state does not hold. The above must have been completed before the within period of the guard ends.

The second Event Assertion rule in line 11 in Listing 1.9 shows the case when an assertion event is expected as a response to a guard event. If an assertion event can be detected before the within period of a guard ends, a pass is emitted for that guard at that time. If no event is detected, a fail is emitted at the end of the within period. If the guard events are very close in time, the same assertion event can satisfy several guards.

The third rule in line 12 in Listing 1.9 shows the case when a state of a particular length is required to start within the specified time. The difference to the rule in line 10 is that the assertion for-period does not need to be finished

before the within period of the guard ends. If the system is already in the asserted state, time is counted from the start of the guard.

4.9 Miscellaneous Modifiers

The statement `ignore` $<$ | $>$ *timeout* tells the evaluation core to ignore any fails before or after the specified time. This is used when there are disturbances at startup or shut down of the system under test. There is also a statement `Allow` *timout* `fail` that makes the G/A ignores fails of the specified length. It is typically used to rule our sampling errors that may lead to fails due to the assertion appears to change before the guard due to sampling errors.

4.10 Timing Considerations

Another core feature of T-EARS, partially present in the early prototypes, is the possibility to specify timing information in logical expressions. Using the keyword `for`, the length of an interval could be specified and using the keyword `within`, timeouts, or grace times could be specified. Further research, however, revealed that longer expressions with the timing keywords applied to each sub-expression were difficult to comprehend or even evaluate correctly. One culprit was that the `for` keyword filtered out intervals longer than the specified timeout, regardless of the expression context, which sometimes yielded very confusing results. The reason is that we, as human readers, have different expectations on what effect `for` has, depending on the expression context. Our solution is to separate different filtering expectations into different keywords. The construction `longer than` and `shorter than` as described in Section 4.5 filters intervals with respect to length, regardless of the current context. Concerning the `for` keyword, the new grammar considers the perception of relative time base: Consider the expressions `R for 10s`. Regarding R as a binary signal, the expression evaluates to one event each time the signal has been true for 10 seconds. At first glance, this seems to be an exact definition, but putting the expression into different contexts reveals some interesting properties.

In the context of a guard expression, we expect (R for 4s) to be "measured" from the start of each interval in R, resulting in a series of events, say $[p_1, p_2, \ldots, p_n]$. In the assertion context, however, there is an implicit assumption that the assertion evaluation is a consequence of a guard event (or interval) and hence, (R for 4s) is expected to be measured from each p_i in the guard expression, i.e., the time for the associated guard. As a consequence of the potential for confusion, the new T-EARS restricts the usage and the meaning of the `for` keyword, as described in Section 4.5 and 4.8. Mixing the `for` keyword with the `within` keyword makes things even more complicated. Further, the effect of `within..for` is different from the effect of `for..within` on an expression.

4.11 General Structure of a T-EARS Test Case

Putting the pieces together, Listing 1.10 illustrates a minor test case:

```
1  //-------------  REQ 558-S1------------
2  // While the vehicle is underway with a speed more thatn 10 km/h,
3  // the doors shall be locked.
4
5  ////////  Abstract G/A   /////////
6  //def intervals MOVING      = [[5s,30s],[100s,300s]]
7  //def intervals DOOR_LOCKED = [[6s,31s],[120s,300s]] // PASS,FAIL
8
9  //// G/A Concretization (Can be centralized in a main def file) /////
10 // System version 1.2.4
11 const DOORS = 1
12 const DOOR_LOCKED_MASK  = 8
13
14 def intervals MOVING =
15     MWT_Standstill == false and MWT_BUS2_Speed > 10
16 def intervals DOOR_LOCKED =
17      bitmask(DOOR_LOCKED_MASK,MWT_door_lock) == DOOR_LOCKED_MASK
18
19 // G/A Definition(s)
20 '558-S1' = where DOORS > 0
21            while MOVING == true
22            shall DOOR_LOCKED == true within 3s
```

Listing 1.10: General Structure of a T-EARS Script

In the example (Listing 1.10), there are three regions of particular interest. The first region (Line 1–7) contains requirement information and example data for an abstract G/A, followed by a region with structural elements (lines 10–17) that make G/As easier to read. These connect the abstract signals to expressions of actual log data. If aliases, constants, and definitions are common to many G/As, they are typically put in a shared file instead. The last region of interest at lines 19-22 is where the actual passive test case is defined. Any number of named G/As can be specified.

5 Discussion on T-EARS Improvement

The early prototypes of the T-EARS language allowed experimenting with a great variety of expressions that could be defined and evaluated against real industrial requirements. Many of these were useful, while others turned out to be less useful. In this work, the goal is to promote useful expressions while suppressing less useful ones and updating the language to be more complete and intuitive. Thus, the goal here concerns updates of the language constructs to improve readability without losing expressiveness. We achieved this goal through a minimal set of top-level boilerplates, introducing strong typing, restricting the usage of timing keywords, and defining new keywords to avoid ambiguous definitions. Although the sum of keywords and language constructs added outgrew the ones removed, the result is a language that more clearly corresponds to the EARS patterns. Instead of automatic conversions between events and intervals, there are now only a few well-defined ways of constructing expressions between the types. One operation that was explicitly removed was (*Event* and *Event*). Since this would require the timestamps to be precisely matching, allowing such an operation would, in practice, only add confusion to the test cases. When the new restricted grammar was applied to old T-EARS expressions, it

revealed quite a few misunderstandings regarding events/intervals. Concerning the expressiveness, the first three EARS patterns (Ubiquitous, State-Drive, and, Event-Driven) are (still) wholly covered by the G/A boilerplates. The fourth EARS pattern (Option) was a strong suggestion from the testers in the case study [9]. The testers wished to have conditionally evaluated G/As depending on configuration information and other G/As. Here rudimentary support was added for Boolean expressions and constants. However, it is still not possible to use the activation or result of a G/A to turn on / off others' evaluations. The fifth EARS pattern can be accomplished but the keywords `if` and `then` may introduce confusion around states or events and are thus not included in T-EARS. The sixth EARS pattern (complex) allows mixing while and when which generated expressions that were inherently difficult to understand. The complex EARS pattern is instead realized in T-EARS using the well defined composition rules of Events and Intervals. Another vital requirement from the testers was user-defined functions. Albeit rather crude, this is now possible in T-EARS since the definitions are evaluated late, and aliases can be used as parameters for the definitions.

Finally, a word about timing. While the early T-EARS prototypes used the `for`-keyword as a filter (keeping all intervals longer than the specified timeout), the tester's expectations differed depending on where the keyword was specified. In some cases, an event was expected after the timeout. In other cases, intervals of precisely the length of the timeout. Further, when the `for` expression was used in an assertion, their semantic meaning was unclear. The remedy was to remove the filtering semantics altogether and introduce keywords for filtering (`longer than, shorter than`), while the `for` keyword was restricted to a few consistent meanings. Further, moving the `within` keyword from individual sub-expressions to the G/A boilerplate reduced confusion. Consider the example: R_1 *for* $10s$ *and* P_1 *within* $2s$ *or* P_2 *within* $4s$. In the example, it is not clear where the time starts. Using the left-hand side of the sequence operator, or the guard as the time base for the timing specifications `within` the evaluation of the timeouts became consistent with the expectations of the testers' intuition.

6 Related Work

This work relates to passive testing, specification of test cases, and the tool support for the use of passive testing and specification of such test cases. We rely on the work of independent guarded assertions [22,12] introduced for increasing the testing parallelism in the vehicular domain. An earlier approach to guarded assertions has been evaluated by Rodriguez Navas et al. [22] and a model-checker has been used for both modelling and test case execution. In this paper, we improve upon this by translating these test cases directly from requirements. Regarding the testability of such requirements, Pudlitz et al. [19,20] used a markup language by relying on annotations of the natural language. Different from this approach, our improved T-EARS language is using a temporal specification of requirements. All these approaches are similar to the passive testing technique as

it is outlined by Cavalli et al. [3] and also relate to run-time verification [13,23]. Since the use of these techniques relies on the formal specification of test cases, several researchers have attempted to use patterns and graphical models for the formalization of both requirements and test cases [2,7,25,1]. The specification of passive testing using the improved T-EARS takes another route by focusing on simplicity and closeness to the requirements text. T-EARS is based on an Easy Approach to Requirements Syntax (EARS), proposed by Rolls-Royce for the creation of semi-structured natural language requirements [16]. EARS has been used in several domains and has been shown to be useful for handling real-world requirements [14,15]. Regarding the tooling for the specification of passive test cases, several researchers have focused on providing support for specification patterns [6,18] and creating monitors and guarded assertions in Matlab [26]. Related to runtime monitoring, Rabiser et al. [21] developed a domain-specific language for defining and checking constraints at runtime.

7 Conclusion and Future Work

We have presented an updated T-EARS language grammar together with a semi-formal specification of the semantics behind the language. The update consists of a careful re-definition of the grammar and semantics for e.g., test case structure and temporal specification. By restricting the possible G/A patterns to a few well-defined boilerplates, the language and its new evaluation core now have a closer correspondence to the EARS patterns. These boilerplates also match the intuition of the testers better. The intuition is also increased by making the notation of timeouts context-dependent, i.e, the guard time domain is now a natural base for the assertion time domain.

Although T-EARS has taken a significant step forward, there are still some features deferred to future research. The first one primarily concerns the accompanied evaluation-core of the T-EARS language. While not necessary from an evaluation point of view, other attempts to create more user-friendly specification languages provide semantic mappings to proven temporal logic such as MITL or LTL. Creating such a mapping for T-EARS would allow certified evaluation tools rather than JavaScript that is used today. Such a mapping would also move the T-EARS evaluation from the offline to the online domain. Another issue is that, while the grammar supports specifying negative time like **when** P_G **shall** P_A **within** $-t$, the current semantics do not. Finally, specifying an evaluation aggregation policy for passive testing is needed to allow a drill-down analysis approach on the increased number of results.

Acknowledgement

The work in this study has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement Nos. 871319, 957212; from the Swedish Innovation Agency (Vinnova) through the XIVT project and from the ECSEL Joint Undertaking (JU) under grant agreement No 101007350.

References

1. Asteasuain, F., Braberman, V.: Specification patterns can be formal and still easy. In: International Conference on Software Engineering and Knowledge Engineering (SEKE'01). pp. 430–436. Knowledge Syst Inst Grad Sch, Knowledge Systems Institute (2010)
2. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. IEEE Transactions on Software Engineering **41**(7), 620–638 (2015)
3. Cavalli, A.R., Higashino, T., Núñez, M.: A survey on formal active and passive testing with applications to the cloud. Annals of telecommunications (3), 85–93 (2015)
4. Daniel, F., Eduard, E., Wasif, A., Daniel, S., Thomas, G., Avenir, K.: From natural language requirements to passive test cases using guarded assertions. In: International Conference on Software Quality, Reliability and Security (QRS'18). pp. 470–481. IEEE Computer Society (2018)
5. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: International Conference on Software Engineering (ICSE'99). pp. 411–420. Association for Computing Machinery (1999). https://doi.org/10.1145/302405.302672
6. Filipovikj, P., Jagerfield, T., Nyberg, M., Rodriguez-Navas, G., Seceleanu, C.: Integrating pattern-based formal requirements specification in an industrial toolchain. In: International Computer Software and Applications Conference (COMPSAC'16). vol. 2, pp. 167–173. IEEE Computer Society (2016)
7. Filipovikj, P., Nyberg, M., Rodriguez-Navas, G.: Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In: International Requirements Engineering Conference (RE'14). pp. 444–450. IEEE Computer Society, Los Alamitos, CA, USA (08 2014). https://doi.org/10.1109/RE.2014.6912296
8. Flemström, D., Gustafsson, T., Kobetski, A.: Saga toolbox: interactive testing of guarded assertions. In: International Conference on Software Testing, Verification and Validation(ICST'17). pp. 516–523. IEEE Computer Society (2017)
9. Flemström, D., Gustafsson, T., Kobetski, A.: A case study of interactive development of passive tests. In: International Workshop on Requirements Engineering and Testing (RET'18). p. 13–20. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3195538.3195544
10. Flemström, D., Gustafsson, T., Kobetski, A., Sundmark, D.: A research roadmap for test design in automated integration testing of vehicular systems. In: International Conference on Fundamentals and Advances in Software Systems Integration (FASSI'16) (2016)
11. Flemström, D., Jonsson, H., Enoiu, E.P., Afzal, W.: Industrial scale passive testing with t-ears. In: Conference on Software Testing, Verification and Validation (ICST'21). pp. 351–361. IEEE Computer Society, Los Alamitos, CA, USA (04 2021). https://doi.org/10.1109/ICST49551.2021.00047
12. Gustafsson, T., Skoglund, M., Kobetski, A., Sundmark, D.: Automotive system testing by independent guarded assertions. In: International Conference on Software Testing, Verification and Validation Workshops (ICSTW'15). pp. 1–7. IEEE Computer Society (2015). https://doi.org/10.1109/ICSTW.2015.7107474
13. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5), 293–303 (2009)

14. Mavin, A., Wilkinson, P.: Big ears (the return of "easy approach to requirements engineering"). In: International Conference on Requirements Engineering (RE'10). pp. 277–282. IEEE Computer Society, Los Alamitos, CA, USA (10 2010). https://doi.org/10.1109/RE.2010.39
15. Mavin, A., Wilksinson, P., Gregory, S., Uusitalo, E.: Listens learned (8 lessons learned applying EARS). In: International Requirements Engineering Conference (RE'16). pp. 276–282. IEEE Computer Society, Los Alamitos, CA, USA (09 2016). https://doi.org/10.1109/RE.2016.38
16. Mavin, A., Wilkinson, P., Harwood, A., Novak, M.: Easy approach to requirements syntax (ears). In: International Requirements Engineering Conference (RE'09). pp. 317–322. IEEE Computer Society (2009)
17. Merz, F., Sinz, C., Post, H., Gorges, T., Kropf, T.: Bridging the gap between test cases and requirements by abstract testing. Innovations in Systems and Software Engineering pp. 1–10 (2015)
18. Miao, W., Wang, X., Liu, S.: A tool for supporting requirements formalization based on specification pattern knowledge. In: International Symposium on Theoretical Aspects of Software Engineering (TASE'15). IEEE Computer Society (2015). https://doi.org/10.1109/TASE.2015.13
19. Pudlitz, F., Brokhausen, F., Vogelsang, A.: What am i testing and where? comparing testing procedures based on lightweight requirements annotations. Empirical Software Engineering **25**(4), 2809–2843 (2020)
20. Pudlitz, F., Vogelsang, A., Brokhausen, F.: A lightweight multilevel markup language for connecting software requirements and simulations. In: International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'19). Springer, Berlin, Heidelberg (2019)
21. Rabiser, R., Thanhofer-Pilisch, J., Vierhauser, M., Grünbacher, P., Egyed, A.: Developing and evolving a dsl-based approach for runtime monitoring of systems of systems. Automated Software Engineering **25**(4), 875–915 (2018)
22. Rodriguez-Navas, G., Kobetski, A., Sundmark, D., Gustafsson, T.: Offline analysis of independent guarded assertions in automotive integration testing. In: International Conference on Embedded Software and Systems (ICESS'15). pp. 1066–1073. IEEE Computer Society (2015). https://doi.org/10.1109/HPCC-CSS-ICESS.2015.251
23. Selyunin, K., Nguyen, T., Bartocci, E., Grosu, R.: Applying runtime monitoring for automotive electronic development. In: International Conference on Runtime Verification (RV'16). pp. 462–469. Springer, Berlin, Heidelberg (2016)
24. Sneed, H.M.: Bridging the concept to implementation gap in software system testing. In: International Conference on Quality Software (QSIC'08). pp. 67–73. IEEE Computer Society, Los Alamitos, CA, USA (08 2008). https://doi.org/10.1109/QSIC.2008.48
25. Walter, B., Hammes, J., Piechotta, M., Rudolph, S.: A formalization method to process structured natural language to logic expressions to detect redundant specification and test statements. In: International Requirements Engineering Conference (RE'17). IEEE Computer Society (2017). https://doi.org/10.1109/RE.2017.38
26. Zander-Nowicka, J., Schieferdecker, I., Marrero Perez, A.: Automotive validation functions for on-line test evaluation of hybrid real-time systems. In: Autotestcon. pp. 799–805. IEEE Computer Society (2006). https://doi.org/10.1109/AUTEST.2006.283767