Offline Analysis of Independent Guarded Assertions in Automotive Integration Testing

Guillermo Rodriguez-Navas
Mälardalen University
Västerås, Sweden
guillermo.rodriguez-navas@mdh.se

Avenir Kobetski and Daniel Sundmark
Swedish Institute of Computer Science
Kista, Sweden
avenir.kobetski@sics.se, daniel.sundmark@sics.se

Thomas Gustafsson
Scania CV AB
Södertälje, Sweden
thomas.gustafsson@scania.com

***Abstract*—The size and complexity of software in automotive systems have increased steadily over the last decades. Modern vehicles typically contain numerous electrical control units (ECUs), and more and more features require real-time interaction between several dedicated ECUs (e.g., gearbox, brake and engine control units) in order to perform their tasks. Since system safety and reliability must not be adversely affected by this increase in complexity, proper quality assurance is a must. Such quality assurance is often performed by testing the system in different levels of integration throughout the development process. However, the growth of complexity of the system under test aslo affects the testing, making it laborious, difficult and costly.**

This paper presents a novel method for efficient offline analysis of traces, which has been especially tailored for integration testing of automotive systems. The method exploits the recently defined concept of independent guarded assertion in order to formally describe the events that are relevant for the analysis as well as the expected behavior in those events. The offline analysis is implemented using a standard commercial model checker and has shown good performance in the conducted experiments.

I. Introduction

The primary means for quality assurance of software-intensive systems in the automotive industry today is by using testing. Testing in software and systems engineering generally and historically suffer from various problems, including being an underprioritized activity [1], being labor-intensive and costly [2], [3], and lacking sufficient tool and infrastructure support [4]. A recent review by Kasoju et al. [5] reveals that testing of automotive systems is no exception, and suffers from most of the problems commonly experienced in general software and system development.

Given their embedded nature, automotive and vehicular systems are typically developed using a traditional V-model of requirements engineering, design, development, integration and testing. At the full-vehicle integration level, the systems under test (SUTs) are generally divided into separate functional parts, which are tested in isolation by exercising scripted scenarios in Hardware-In-the-Loop (HIL) test rigs or Software-In-The-Loop (SIL) type simulators.

While practically useful and intuitive, this type of testing suffers from a number of drawbacks. Specifically, since test scenarios are hard-coded, there is little variation in how the testing is performed over time. In addition, such test scenarios are typically based on requirements defining how the system should behave during normal operation, potentially overlooking less intuitive scenarios and non-specified interaction sequences. Moreover, in the case of HIL testing, execution of only one scenario at a time in a batch run causes poor utilization of expensive bottleneck equipment.

In order to address the above problems, the notion of Independent Guarded Assertions (IGAs) was recently introduced as an alternative approach for automated system-level testing of automotive systems [6]. When using IGAs, the state-changing stimuli (i.e., the inputs) for each test case are separated from the verdict-generating assertions (i.e., the comparisons between the actual and expected output).

The assertions are then guarded in such a way that they only evaluate the system response when they recognize a specific sequence of events occurring in the system under test. Independent guarded assertions allow for 1) parallelized execution of test cases (both in terms of guarded assertions and sequences of stimuli), 2) repeated and frequent evaluation of assertions, often with the system being in states not explicitly considered in the original requirements, and 3) reduction of testing time, since repetitive and time-consuming test case setup and cleanup can be reduced if not ignored.

While independent guarded assertions provide a number of benefits over traditional scenario-based testing, the online nature of IGAs (and specifically the IGA guards) may incur probe effects on the system under test and the test framework. These probe effects hamper the scalability of the method and put limits on the number of parallel IGAs that can be active at a given point in time during testing. Further, when using online versions of IGAs, the drawback of having to utilize the bottleneck HIL infrastructure for all testing is still a potential impediment for efficient and effective testing.

In this paper, we describe an alternative method for testing independent guarded assertions using a commercial model checker and recorded system traces. This offline approach, as contrasted to the previously proposed online variant of IGA-based testing, provides several benefits:

- The use of offline traces eliminates probe effects in the analysis, and the online real-time limitation on the number of IGAs active during testing becomes irrelevant.
- Test cases can be run as soon as a set of guards is valid, since the requirement of access to the bottleneck test equipment is eliminated.

- There is a clear separation of concerns between the IGAs and the verification engine (the model checker), which facilitates review and validation of the tests by external experts.
- The approach can be used off-site for analysis, continuous improvement, and research purposes, e.g. to elaborate on how to optimize test design (including design of test stimuli sequences, guards, and assertions), or how to perform the actual verification in an efficient way.

For the model-checking-based offline trace analysis, we use the UPPAAL model checker [7]. In addition to providing an in-depth description of the proposed method, and how the IGAs can be represented as UPPAAL timed automata, we also provide an explanatory and evaluative case study performed at the full-vehicle integration testing level at the Swedish truck manufacturer Scania CV[1].

The remainder of this paper is structured as follows: Section II provides a background on automotive system testing, independent guarded assertions, and UPPAAL. Section III describes our proposed approach for offline analysis of IGAs. Section IV presents an explanatory and evaluative case study of the proposed method. In Section V, related work is summarized, and Section VI concludes the paper and discusses potential directions for future work.

II. BACKGROUND

This paper expands on existing work on *integration testing of automotive systems* by *independent guarded assertions*. The offline analysis of integration test traces is performed with UPPAAL, a state-of-the-art model checker. The notions required for understanding this approach are reviewed here.

A. Automotive System Integration Testing

In our experience from working with several OEMs in the automotive and vehicular domains, automotive system development typically follows a fairly traditional V-model of development, integration and testing. At full-vehicle integration level, in addition to in-vehicle testing, the systems under test (SUTs) are generally divided into separate functional parts, which are tested in isolation by defining use-case based test sequences. This type of test design, while not limited to embedded system testing, is commonly referred to as scenario-based testing [8]. Scenario-based testing, using hard-coded scripted test cases seems to be the current de facto standard of control system testing in the automotive, as well as in the more general vehicular industry. Often times, these test cases are executed on the system under test by means of hardware-in-the-loop (HIL) or software-in-the-loop (SIL) based integration testing platforms. There is also an increasing trend of utilizing model-based testing for this purpose, and techniques focusing on model-based testing dominate research in the (relatively sparsely populated [9]) automotive testing area (see e.g., [10] and [11]).

[1]www.scania.com

Although scenario-based testing does ensure that certain requirements are covered during testing, it has a number of drawbacks. When using HIL testing, the execution of test cases is time consuming due to the sheer number of test cases executed, and thus often conducted in overnight batch runs. Moreover, test cases are designed using a divide-and-conquer-based approach following the division of the system requirements into smaller functional entities (which is in accordance with recent textbook guidelines for functional test design [12], [13]). Once having been created, scripted and incorporated in the test suite, test cases are typically repetitively executed without much variation. This leads to a situation where only a small portion of the vast set of possible scenarios that the system could be subjected to are thoroughly tested, while the others are left entirely unexplored. In addition, test cases are typically based on requirements defining how the system should behave during normal operation [14]. While this provides valuable confirmation with respect to the system's fitness for use in the normal case, there are results indicating that focusing on normal requirement-based cases might not be the best strategy when trying to maximize fault-detection (see e.g., [15] and [16]).

It should however be noted that scenario-based testing can be seen as a special case of the more general model-based testing approach [8], [17]. Both techniques rely on a divide-and-conquer procedure, where requirements and specifications are broken down into smaller units, either from an architectural or a functional perspective. These smaller units are then expressed as test models (typically some form of state charts) or specific scenarios (that could theoretically be seen as individual paths through the state chart). Such model or artefacts are valuable for testing of the individual units, but typically do not express the expected behavior stemming from interaction between the modeled units.

B. Independent Guarded Assertions

Partially based on previous work on *declarative testing* [18], [19], we recently proposed the use of Independent Guarded Assertions (IGAs) for integration testing [6]. Independent guarded assertions allow for 1) parallellized execution of test cases (both in terms of guarded assertions and sequences of stimuli), 2) repeated and frequent evaluation of assertions, often with the system being in states not explicitly considered in the original requirements, and 3) reduction of testing time, since repetitive and time-consuming test case setup and cleanup can be reduced if not ignored. By 1) and 2), IGA testing inherently and indirectly addresses the lack of focus on interaction between modeled units that is a consequence of divide-and-conquer based approaches such as model- and scenario-based testing.

Testing using IGAs relies on two kinds of inputs: system traces and guarded assertions. A *system trace* is a time-stamped record of the system's state evolution. It contains sampled values of relevant signals observed periodically inside the Hardware In the Loop system (HIL), and collected by the test automation framework. A typical setup is illustrated

in Figure 1, which shows the different functional blocks of the framework for full-vehicle integration regression testing used by Scania, the case company studied in this paper. The sequence of test stimuli is introduced (left hand side of the picture) in order to induce changes of the System Under Test (SUT). Entities in the real-time model and communication buses are stored in the system trace file by the test automation framework (right-hand side of picture), resulting in an ordered and timestamped system test trace bearing information on the state changes and actuator signals resulting from the stimuli sequence used to test the SUT.

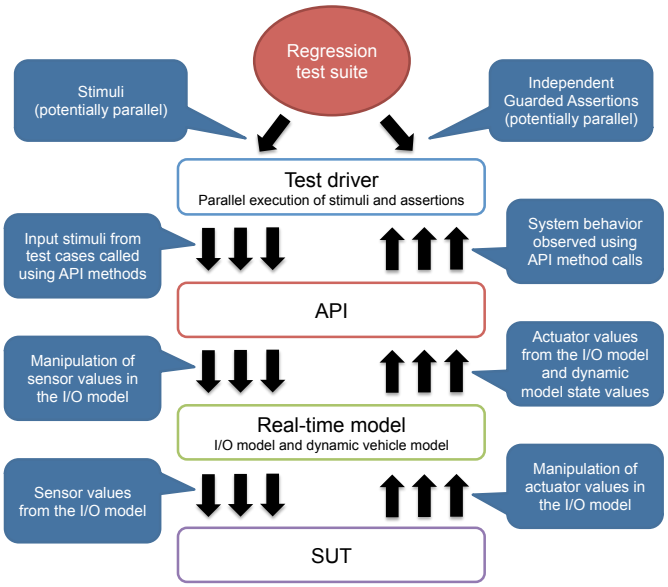

Fig. 1. Independent Guarded Assertions Test Framework.

Even if system traces are most typically collected by executing pre-defined stimuli in a test framework like the one in Figure 1, they could alternatively be recorded during a drive session in a real vehicle, or even using some simulation tool. In the latter cases though, the number of actual signals included into the trace and, in the case of simulation, the extent they reliably represent realistic real-time operation data, could potentially be reduced. In any case, an important condition is that the sampling period of any trace must guarantee that each signal change caused by the input stimuli is detected, thus ensuring that all relevant states and events are stored for further analysis.

A *guarded assertion* represents certain requirements on how the system should behave, and contains two elements: the guard and the assertion. A *guard* consists of a sequence or composition of temporal logic expressions on system signals and describes the conditions (or, more precisely, sequence of events leading to a state) under which a given test is supposed to be assertable (i.e. it can be evaluated). An *assertion* is simply a validation of certain signals with respect to their expected values, which must be satisfied once the corresponding guard holds. Guarded assertions are denoted as G $\Rightarrow$ A. For explanatory purposes, we below provide three examples taken from the automotive industry:

1) $\{gear = reverse \Rightarrow reverse\_light = On\}$
   The reverse light is on when the gear is in reverse.
2) $\{gear <> reverse \Rightarrow reverse\_light = Off\}$
   The reverse light is off when the gear is not in reverse.
3) $\{hazard = On \wedge dir = Off \Rightarrow dir\_ind\ flashing\}$
   Direction indicators shall flash if hazard warning is on, even if the direction indicator button (dir) is off.

Assertions do not have to be necessarily evaluated at one time instant. For example, a test might check how many times a given signal changes its value after a certain state has been reached.

Typically, it is important to validate that a test never fails. This means that when a guard is evaluated to true, its assertion should never evaluate to false. Expressed in temporal logics, it should be verified that *G always implies A*. In our setting, where stimuli are decoupled from the actual tests, it is vital to also check whether a given test has been asserted at all, in order to avoid the trivial fulfillment of the implication when the antecedent is never true, also known as *vacuity problem* [20]. For that, it is enough to validate that the assertable state (G) has been reached at least once.

C. UPPAAL

Model checking is a technique that, given a formal model of a system and a set of properties to be fulfilled by the system, automatically determines whether the possible behavior of said model agrees with the stated properties or not [21], [22]. While the system model is generally specified with some kind of automata, the properties to be verified are usually expressed in some form of temporal logics, such as LTL, CTL or TCTL.

UPPAAL is a model checker based on the theory of *timed automata*, which is particularly suitable for the formal verification of real-time systems [7]. Timed automata extend the traditional concept of finite state automata with a symbolic representation of time, called clock variables, which are used in order to model time progression explicitly. Other arguments for using UPPAAL include separation of concerns between the system model and the tests, good user support and availability of the tools, traceability of the tests (via counter examples), confidence on the correctness of the proofs, and efficiency. Note however that while our framework has been tailored for UPPAAL, other model checkers can be used.

Each timed automaton in UPPAAL contains a number of *locations* and transitions among locations (called *edges*). The state of a timed automaton is the combination of the location together with the value of all the variables, including integer and boolean variables, as well as clocks. Figure 2 shows a model containing two locations and one edge between them.

Over each edge, the defined system variables can be updated, with one restriction: clock variables can only be restarted, e.g., $x = 0$ in Figure 2

Moreover, two or more actions in different timed automata can be synchronized by defining *synchronous channels* in the UPPAAL model (sync! in Figure 2 is synchronized with other edges labeled with sync?). Whenever one transition is

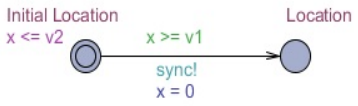

Fig. 2. UPPAAL example

performed, the other transitions, which are synchronized with that, are fired to a new location in the system. A location can be changed in a timed automaton (for both synchronized and regular transitions) whenever the *guard condition* defined for that transition is satisfied. The guard condition can be defined based on an integer, boolean or clock variable, e.g, $x >= v1$ in Figure 2. In order to guarantee progression of the system, for each location an *invariant* can be defined upon the clock variable, which limits the amount of time the model can stay in that location, e.g., $x \leq v2$, where $v2 \geq v1$.

A system can stay in a certain location as long as the edge guard and the invariant of the location are not satisfied. However, a particular type of location, namely *committed location*, is available such that the system should immediately leave this location. This location can interleave with other committed locations, only, which allows us to model atomic actions.

III. Method

The goal of our offline analysis method is to verify that a number of predefined guarded assertions are detected and satisfied within a recorded system trace. This section describes how the concept of model checking can be exploited in order to perform this analysis more efficiently.

A. Verification approach

Figure 3 depicts the three phases in which this method is executed. As indicated in Section II-B, the system traces and the guarded assertions constitute the main inputs of our framework. We assume that prior to the analysis, the system traces have been divided into individual signal traces recorded as Matlab files (.mat). This format is chosen because it is a de-facto standard for the design of control systems in the automotive industry. In the first phase of the analysis, each individual signal trace is transformed into a so-called *Trace automaton*. The rules of this automated transformation will be described in Section III-B .

In the second phase, each guarded assertion is manually decomposed into a number of checks that are also specified as timed automata, which will be called *Guard observers*, and into two additional properties expressed in TCTL, the assertion checks. The principles applied for this decomposition will be discussed in Section III-C

Finally, all the generated automata are combined into a single network of timed automata that is fed to the UPPAAL model checker, whereas the assertion checks are expressed as UPPAAL queries. The results provided by UPPAAL will indicate for each Independent Guarded Assertion whether the

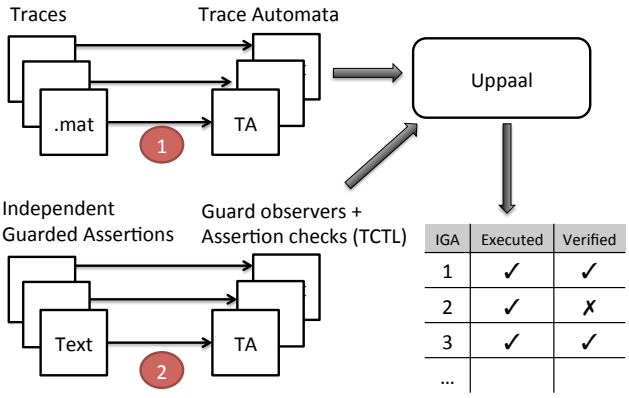

Fig. 3. Proposed framework for offline analysis.

test has been executed or not, and, if executed, whether the test has been verified or not.

B. Automated generation of trace automata

Definition 3.1: **Signal Trace**. A signal trace, denoted V_n, is the discretized version of a continuous system signal V(t), sampled with frequency $f = \frac{1}{T}$, and annotated with the time instant in which each sample was recorded. Thus, V_n is a finite sequence of pairwise values (t_i, v_i), such that $v_i = V(t_i)$; where t_i is called the sampling instant of the i-th sample, and v_i is the sampled value. The hypothesis of periodic sampling is not required specifically for the presented framework, but since it is a common characteristic of most instrumentation environments, it will be assumed without losing generality.

Definition 3.2: **Signal Transition**. Given a signal trace V_n, we say that a signal transition occurs in sample i (or equivalently, at time instant t_i) if and only if $v_i \neq v_{i-1}$. This will be denoted as $st(i) = 1$. By default, $st(0) = 1$.

Signal transitions are very relevant from a verification point of view, because they indicate moments in which the system's state changes. For this reason, they are extracted into a so-called transition vector, which will constitute the basis for constructing the trace automata.

Definition 3.3: **Transition vector**. Given a signal trace V_n, its transition vector $V_n^{\uparrow}$ is a finite sequence of pairs $(t_j^{\uparrow}, v_j^{\uparrow})$ containing only the signal transitions of V_n; it is: $V_n^{\uparrow} = \{(t_i, v_i) \,|\, st(i) = 1\}$. Each value $v_j^{\uparrow}$ is called a *transition value*, while each instant $t_j^{\uparrow}$ is called a *transition instant*.

Definition 3.4: **Trace automata**. The trace automata of a certain signal trace V_n, denoted as A_V, is a deterministic timed automata that accepts the word corresponding to V_n.

There is a straightforward method for creating a trace automata from any signal trace. This trace automata is a timed automata containing one location per each transition value in $V_n^{\uparrow}$, a clock t that is never reset again after initialization of the automaton, and a variable v representing the signal value, which is initialized to $v_0^{\uparrow}$. The transitions between states are specified with one guarded edge connecting every location l_i to location l_{i+1}, with a *guard* $t == t_{i+1}^{\uparrow}$, and the *update* $v = t_{i+1}^{\uparrow}$. To ensure that the state changes happen in the

right moments of time, every location l_i is annotated with the invariant $t \leq t^{\uparrow}_{i+1}$, except for the last location, in which the invariant is $t \leq$ *max. duration of the trace*. Figure 4 shows a simple example of such an automata, where the variables v[i] and t[i] represent the transition values and the transitions instants, respectively.

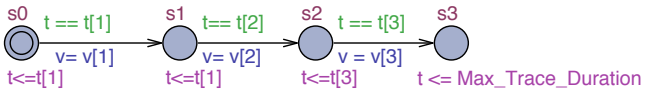

Fig. 4. Exemple of a simple *trace automata* generated with UPPAAL

There are more compact ways to construct an equivalent trace automata, and some of them have been implemented in our framework for evaluation purposes. For instance, it is possible to create a timed automaton that contains one location per each *different* transition value, so that locations can be revisited during the trace. But for the sake of brevity and because they have not been fully assessed, these alternative automata will not be described here.

C. Transformation of Guarded Assertions into timed automata

A Guarded Assertion (GA) consists of a number of condition checks (the *Guard*) that indicate when the system has reached an assertable state; it is, a state in which the *assertion* can be evaluated. The strategy followed in our framework is to define one timed automaton, the *guard observer*, for monitoring the signals of interest for each GA. This observer automaton contains a minimum of four locations: the initial location (wait), in which the Guard is not satisfied, a location (eval) in which the Guard is satisfied and hence the assertion can be evaluated, and two locations (pass and fail) corresponding to the two possible outcomes of the assertion test.

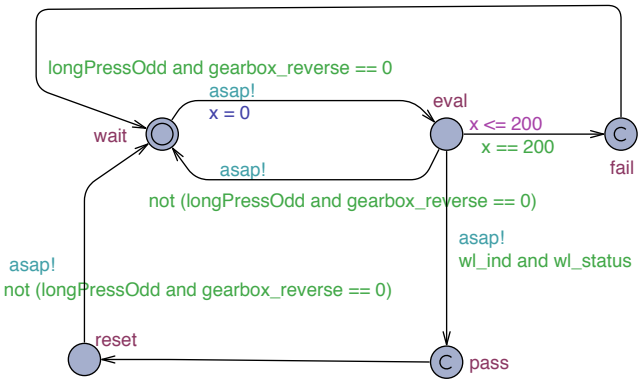

Fig. 5. Example of a Guard observer.

Figure 5 provides an illustrative example of Guard observer, taken from the automotive industry. The goal of this test is to check that: if Button is pressed long an odd number of times, while the gearbox is not in reverse, then both the work light indicator and the work light status must be turned on before 200ms. Note that the transition from location wait to location eval is guarded by the expression *longPressOdd and gearbox_reverse == 0*, corresponding to the Guard. From location eval, the automaton passes the test (i.e. it goes to location pass) only if the assertion *wl_ind and wl_status and x <200* is fulfilled. The test fails (the observer goes to location fail) if the timeout expires without the activation of signals wl_ind and wl_status.

With this setup, the properties to be verified by UPPAAL are:

1) A[] not pass. This property should be *falsified*, indicating that there is at least one case in which the Guard was true and the test was verified.
2) A[] not fail. This property should be *verified*, indicating that there is no case in which the Guard was true and the Assertion was false.

It is important to remark that whenever the Guard is defined over complex combinations of inputs, the system modeler must define additional automata describing the system state as required. For example, for the observer of Figure 5, the modeler had to define an automaton for detecting short and long presses of the bottom, and an automaton for discriminating between odd and even number of presses. The former automaton is depicted in Figure 6, whereas the latter is shown in Figure 7.

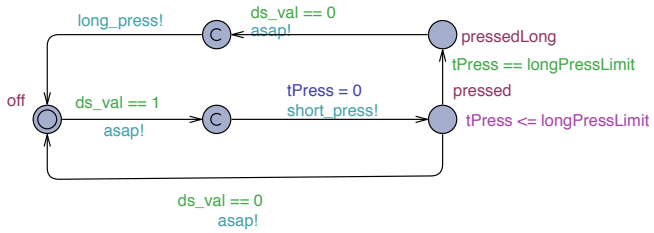

Fig. 6. Automaton for discrimination of type of button press.

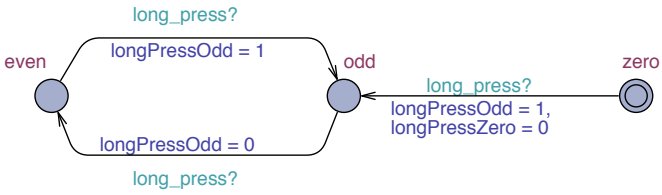

Fig. 7. Automaton for detection of number of long button press.

For more complex IGA, the observer should include intermediate locations between location wait and location eval. We show an example in Figure 8. Note that this automaton includes two locations (preshutdown and preeval) which represent intermediate states of the Guard. The goal of this IGA is to test that once the Guard is reached, the work light indicator flashes intermittently. To detect that, the observer checks whether the work light indicator toggles value 4 or more times within a predefined time interval *to*. However, before the Guard is reached, certain preliminary states need to be traversed, and that is the reason why extra locations are added.

The construction of the Guard observers and the decision of which additional automata to include is still a manual activity,

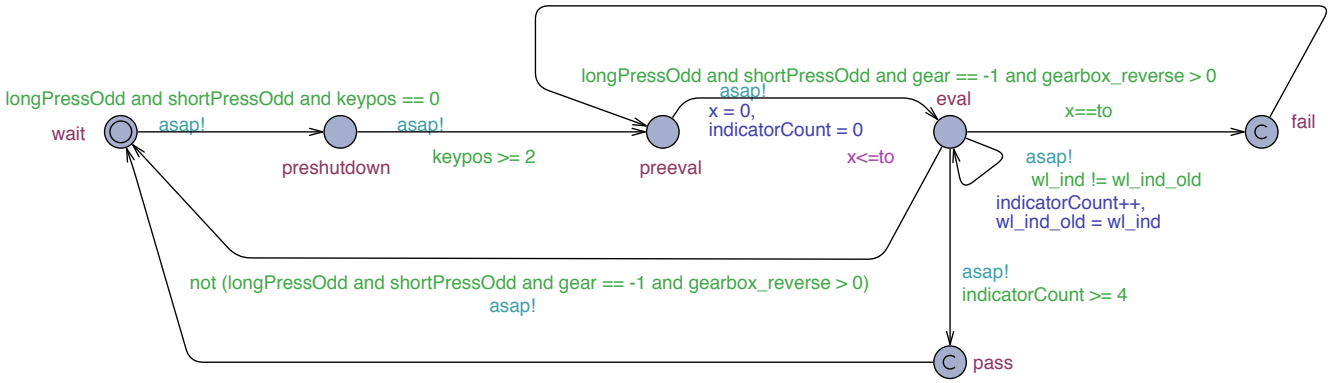

Fig. 8. A Guard observer including more intermediate states and conditions.

which requires good understanding and expertise on the timed automata formalism. However, given the repetitive nature of the tests, we have noticed that some patterns appear quite frequently and with little variation. Due to this, it can be said that the construction of the observers is a semiautomated process. But for improving the usability of the tool, we need to investigate approaches that will allow full automatization.

In contrast to hard-coded script testing, model checking provides separation of concerns between the tests to be checked and the verification engine itself. This offers different opportunities but also has some associated challenges. One interesting opportunity is that more advanced analysis techniques can be applied, for instance for optimization and coverage analysis of the guards. The main challenge is to ensure that the models are correctly specified and accurately represent the designer's intent.

IV. Case Study

This section describes a case study where Scania's test automation framework is used as a baseline and compared to tests executed using Uppaal.

A. Study Object: Scania's Test Automation Framework

Scania is using a test automation framework for its automatic regression testing that is based on a client/server architecture. Each test script is implemented using a test script interface. This interface is the client side. The server side is implemented as a set of servers, and they constitute the core functionality needed for performing the tests. One of the servers handles the communication between a PC and the HIL. Each entity that is readable and/or writable in the HIL has a path. The server can support two calls: `get_value` and `set_value`, which operate on the path.

By replacing the above mentioned server, it is possible to instead take the values from a trace file. This is implemented as follows:

- The server knows the number of clients connected to it.
- The server keeps a current time denoted t, where $t = 0$ is the start of the trace.
- The server keeps an ordered list of requested events. When all clients have made a request, then the server takes the closest future event, at t', and releases the client that made the request, and sets $t = t'$.
- Each call to `set_value(p,v)` where `p` is a path and `v` the value it is set to, finds a time $t'' \geq t$ in the trace where the path has the value, and inserts an event at t''.
- Each call to `get_value` reads the value at t and returns it.
- Each call to `sleep(s)` inserts an event at $t + s$. This means that the client side shall use a sleep method in the server side.

The above implementation leads to the server synchronizing all clients to a common time base. The test scripts being considered are expressed as independent guarded assertions, which means that they do not contain any stimuli, so no `set_value` calls are used, only `get_value` calls. This means that the clients need to poll the HIL for state changes. If no state changes shall be missed by the test script, then the sampling interval must be the same as the recording sampling interval.

B. Experiment with scripts

Several of Scania's test scripts have been transformed into Independent Guarded Assertions (IGAs) [6]. We report the execution time of some of these IGAs using Scania's test automation framework, described in Section IV-A, in Table I. The trace is around 495 seconds long, i.e., 8 minutes and 15 seconds, and it contains stimuli for a number of different test scripts (see [6] for details). The stimuli represent the order of these test scripts executed one after another. The number of times assertions that will be executed depends on how general the IGAs can be made, and the stimuli itself. In Table I, we see that script 1 could only perform assertions once, since it tests the direction indiciator in the event of hazard warning lights during engine off, and this situation only occured one time in the stimuli, whereas for script 3 the assertions were performed many more times. This script instead tests that the worklight function is not activated if the function is turned off or in a special mode and the gear is neutral or forward. This situation occurs more often in the stimuli, and the tests can be performed 8144 times. The sampling interval is 10 ms, so

TABLE I
EXECUTION TIMES OF IGAS EXECUTED BY SCANIA'S TEST AUTOMATION FRAMEWORK AGAINST A TRACE FILE.

Script	Start	End	Exec. Time	# tests
1	12:52:07	12:55:55	3min 48 sec	1
2	12:56:16	12:59:55	3min 39 sec	2
3	12:19:36	12:29:02	9min 26 sec	8144
4	13:02:57	13:10:18	7min 21 sec	2778

there are $495 \times 100 = 495000$ samples where a test can be performed.

The implementation of IGA can require multiple concurrent threads, depending on the complexity of the Guard. Script 1 and 2 use only one single thread, whereas script 3 and 4 use three concurrent threads (one thread counts the number of times a button has been pressed, one thread checks the state of the gear selector, and the third fuses this information with other guards into a decision whether to test or not). We see that the execution time depends heavily on the number of threads. The reason is the implementation of the test automation framework server, where the threads are synchronized at each sampling point, i.e., every 0.01 seconds, as described in Section IV-A, so for script 1 and 2 the server can immediately decide to release the single thread on a request for a future event, but script 3 and 4 must wait for all three threads to request a future event.

In summary, IGAs can be executed, without any adaptations, against trace files, but the execution time depends heavily on the implementation of the guards, and Table I shows execution times ranging from 44% to 114% of real-time (219 sec / 495 sec and 566 sec / 495 sec, respectively).

C. Experiments with the model checker

The traces and IGAs of the previous experiment were used for evaluating the performance of our offline analysis method. Despite the initial difficulties for expressing the Guards as observers, the results are very promising. Once the model is built, the time required for evaluation of a test is around 0.25s, which is several orders of magnitude lower than the online method.

Even if further investigation is needed for performance evaluations, these preliminary results show that model checking is suitable for integration testing of functions. It will be interesting to investigate the length of the traces that can be processed, but apparently the well known problem of state space explosion does not seem a limiting factor for this type of analysis in automotive integration testing.

V. RELATED WORK

Offline analysis of traces for temporal logic properties is related to several previously proposed approaches for runtime monitoring. In [23], Delgado et al. propose a taxonomy and a catalog of runtime software-fault monitoring tools. A partial classification of the method proposed in this paper according to this taxonomy would state that we describe an *automata-based asynchronous offline* monitoring approach with an *event-based* monitoring directive.

According to the authors of the above taxonomy and catalog of runtime monitoring tools, the only existing method for trace analysis with a similar classification (at the time of the review) is Monitoring-oriented Programming (MoP) [24] (see also [25]). However, MoP is a highly generic concept rather than a specific method in itself. Specifically, it does not enforce any limitations on the formalism used to express the desired properties of the system under observation/test (as long as a translator exists to transform the formalisms to something the monitoring technique can make use of). Neither is it limited to only asynchronous offline monitoring, but can also be used synchronously offline, as well as online. It is consequently not tailored specifically for timed-automata based properties and temporal logic-type queries, and provides no such support unless a translator is being developed spefically for this purpose.

More recently, Bauer et al. [26] describe the use of runtime verification of properties expressed in lineartime temporal logic (LTL) or timed lineartime temporal logic (TLTL). Sulzmann and Zechner [27] study finite trace matching algorithms using LTL. They use a constructive algorithm for constructing proofs from the LTL formulas, to give more details why a proof is matched.

In the automotive field, Zander-Nowicka *et al.* have used *automotive validation functions* for testing models used to realize functions in a vehicle [28]. An automotive validation function has preconditions and makes assertions if the preconditons are true. An automotive validation function is thus a guarded command. In their approach, several automotive validation functions were used for testing one specific function, but in our work we aim at concurrently testing several functions.

In summary, a lot of work has gone into using runtime verification, and a fair share of this work has focused on offline trace analysis and checking of temporal logic properties. However, to the best of our knowledge, research work studying the practical use of offline trace analysis on real systems in industrial contexts is more scarce. The method, and the application of the method, described in this paper contributes to this gap in the current body of knowledge.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we have investigated how model checking can be utilized for performing integration testing of funtions in a complete vehicle electrical system. Our work shows how to perform offline guarded assertion analysis of recorded traces using a model checker. We show that it is possible to automatically transform the system trace, i.e., the periodic samples of entities in the real-time model of a HIL, into a timed automaton. Furthermore, we discuss how guarded assertions are manually transformed into timed automata. Properties of these automata can be checked by a model checker. In our work, we have used UPPAAL for this purpose because of the explicit representation of time provided by this tool, which is very adequate for real-time requirements.

Early experimental results from the full-vehicle integration testing at Scania indicate that this technique may result in

significantly faster run-times than executing the existing test scripts against the system trace. Despite these promising results, the question of whether our approach can scale up to more complex scenarios still remains open. Extremely large traces and complex guarded assertions may generate state space problems. On the other hand, the offline trace analysis is linear in the sense that the state space exploration is driven by and inherently constrained by the recorded trace to be analysed. Regardless, further investigation is required in order to assess the limits of the model-checking based approach.

Regarding future work, we envision a number of directions for future research on offline analysis of independed guarded assertions:

- **Larger-scale evaluation:** While the case study presented in this paper provides an explanation of how offline IGA analysis works in an industrial context, and an early evaluation of the potential benefits of the approach, it is too limited for providing any reliable and more generally valid evidence of these effects. More rigorous and larger scale experimentation is required for this purpose, preferably evaluating the approach in different industrial settings.
- **Optimization-based test sequence generation:** Through the frequently executed assertions, IGAs provide a more thorough evaluation of the system under test, but they do not by themselves provide an increased coverage in terms of the fraction of state space exercised during testing. Given that an infrastructure of IGAs is in place, the design or generation of test stimuli sequences is still an open issue. For this purpose, optimization-based test sequence generation (e.g., by using heuristics or meta-heuristic techniques) can be explored. Potential objectives for such searches may include minimization of test time, maximizing state space coverage, finding test sequences with the maximum diversity, or deriving test sequences that invoke as many IGA assertions as possible.
- **Appropriate representation of IGAs:** While the UPPAAL-based approach of modeling IGAs presented in this paper is seemingly efficient and effective for the purpose of offline trace analysis, it may not be the most intuitive and appropriate format for automotive system test engineers to work with. Alternative formats for temporal (and probabilistic) requirements, that are more similar to natural language representations, have been proposed by e.g., [29], [30]. Such representations, if translatable to UPPAAL-automata, could make the approach more accessible. Our first investigations on this direction have shown the adequacy of this approach for the automotive domain [31].

ACKNOWLEDGMENT

This work has been partially funded by VINNOVA, Sweden innovation agency, within the FFI program projects VeriSpec and PINT, and by the Swedish Knowledge Foundation (KKS) within the Prospekt project SaDIES.

REFERENCES

[1] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist, "Technical debt in test automation," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, April 2012, pp. 887–892.

[2] V. Garousi and J. Zhi, "A survey of software testing practices in canada," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1354 – 1376, 2013.

[3] S. Ng, T. Murnane, K. Reed, D. Grant, and T. Chen, "A preliminary survey on software testing practices in australia," in *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, 2004, pp. 116–125.

[4] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Tech. Rep. 7007, 2002.

[5] A. Kasoju, K. Petersen, and M. V. Mäntylä, "Analyzing an automotive testing process with evidence-based software engineering," *Information and Software Technology*, vol. 55, no. 7, pp. 1237 – 1259, 2013.

[6] T. Gustafsson, M. Skoglund, A. Kobetski, and D. Sundmark, "Automotive System Testing by Independent Guarded Assertions," proceedings of the 10th Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART), April 2015.

[7] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[8] A. Bertolino, E. Marchetti, and H. Muccini, "Introducing a reasonably complete and coherent approach for model-based testing," *Electr. Notes Theor. Comput. Sci.*, vol. 116, pp. 85–97, 2005.

[9] E. Bringmann and A. Krämer, "Model-based testing of automotive systems," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ser. ICST '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 485–493. [Online]. Available: http://dx.doi.org/10.1109/ICST.2008.45

[10] G. Park, D. Ku, S. Lee, W.-J. Won, and W. Jung, "Test methods of the autosar application software components," in *ICCAS-SICE, 2009*, Aug 2009, pp. 2601–2606.

[11] A. Ray, I. Morschhaeuser, C. Ackermann, R. Cleaveland, C. Shelton, and C. Martin, "Validating automotive control software using instrumentation-based verification," in *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, Nov 2009, pp. 15–25.

[12] M. Young and M. Pezze, *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005.

[13] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge University Press, 2008.

[14] L. M. Leventhal, B. Teasley, D. S. Rohlman, and K. Instone, "Positive test bias in software testing among professionals: A review," in *Selected papers from the Third International Conference on Human-Computer Interaction*, ser. EWHCI '93. London, UK, UK: Springer-Verlag, 1993, pp. 210–218. [Online]. Available: http://dl.acm.org/citation.cfm?id=646181.682601

[15] L. M. Leventhal, B. E. Teasley, and D. S. Rohlman, "Analyses of factors related to positive test bias in software testing," *Int. J. Hum.-Comput. Stud.*, vol. 41, no. 5, pp. 717–749, Nov. 1994. [Online]. Available: http://dx.doi.org/10.1006/ijhc.1994.1079

[16] A. Causevic, R. Shukla, S. Punnekkat, and D. Sundmark, "Effects of negative testing on tdd: An industrial experiment," in *International Conference on Agile Software Development, XP2013*, H.Baumeister and B. Weber, Eds. Springer, June 2013. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=3276

[17] G. Hamon, L. de Moura, and J. Rushby, "Generating efficient test sets with a model checker," in *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, Sept 2004, pp. 261–270.

[18] P. Alvaro, A. Hutchinson, N. Conway, W. R. Marczak, and J. M. Hellerstein, "Bloomunit: Declarative testing for distributed programs," in *Proceedings of the Fifth International Workshop on Testing Database Systems*. ACM, 2012, p. 1.

[19] E. Triou, Z. Abbas, and S. Kothapalle, "Declarative testing: A paradigm for testing software applications," in *Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference on*. IEEE, 2009, pp. 769–773.

[20] O. Kupferman, "Sanity checks in formal verification," in *CONCUR 2006*, 2006, pp. 37–51.

[21] J.-P. Katoen, "Concepts, algorithms, and tools for model checking," Lecture Notes of the course Mechanised Validation of Parallel Systems, 1998.

[22] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 2001.

[23] N. Delgado, A. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *Software Engineering, IEEE Transactions on*, vol. 30, no. 12, pp. 859–872, Dec 2004.

[24] F. Chen and G. Roşu, "Towards monitoring-oriented programming: A paradigm combining specification and implementation," vol. 89, no. 2, 2003, pp. 108 – 127, {RV} '2003, Run-time Verification (Satellite Workshop of {CAV} '03). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066104810454

[25] ——, "Mop: An efficient and generic runtime verification framework," in *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 569–588. [Online]. Available: http://doi.acm.org/10.1145/1297027.1297069

[26] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 4, p. 14, 2011.

[27] M. Sulzmann and A. Zechner, "Constructive finite trace analysis with linear temporal logic," in *Proceedings of the 6th International Conference on Tests and Proofs*, ser. TAP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 132–148.

[28] J. Zander-Nowicka, I. Schieferdecker, and A. Marrero Perez, "Automotive validation functions for on-line test evaluation of hybrid real-time systems," in *Autotestcon, 2006 IEEE*, Sept 2006, pp. 799–805.

[29] S. Konrad and B. H. C. Cheng, "Real-time specification patterns," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 372–381. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062526

[30] 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," *Transactions on Software Engineering*, 2015.

[31] 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.