Automotive System Testing by Independent Guarded Assertions

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

Mats Skoglund, Avenir Kobetski, Daniel Sundmark
Swedish Institute of Computer Science
Kista, Sweden
firstname.lastname@sics.se

***Abstract*—Testing is a key activity in industry to verify and validate products before they reach end customers. In hardware-in-the-loop system-level verification of automotive systems, testing is often performed using sequential execution of test scripts, each containing a mix of stimuli and assertions.**

In this paper, we propose and study an alternative approach for automated system-level testing automotive systems. In our approach, assertion-only test scripts and one (or several) stimuli-only script(s), execute concurrently on the test driver. By separating the stimuli from the assertions, with each assertion independently determining when the system under test shall be verified, we seek to achieve three things: 1) tests that better represent real-world handling of the product, 2) reduced test execution time, and 3) increased defect detection. In addition to describing our proposed approach in detail, we provide experimental results from an industrial case study evaluating the approach in an automotive system test environment.

I. Introduction

The primary means for quality assurance of software-intensive systems in the automotive industry today is by using testing. At 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 exercising pre-defined (often scripted) test scenarios in Hardware-In-the-Loop (HIL) test rigs.

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 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 [1], [2]). 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 [3]. 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., [4] and [5]).

In this paper, we introduce the use of *independent guarded assertions* in order to reduce testing time, increase defect detection, and improve the real-world representativeness of HIL-based automotive integration testing. More specifically, for each test case, the state-changing stimuli (i.e., the inputs) are separated from the verdict-generating assertions (i.e., the comparisons between the actual and expected output). Then, the assertions are guarded from evaluation based on the state of the system under test, while the sequences of state-changing stimuli are fed to the system under test independently and with very few restrictions. The assertion guards merely monitor the state of the system during test execution. Each instance where the system state satisfies the conditions of an assertion guard, the assertion is performed and a verdict generated.

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 repetetive and time-consuming test case setup and cleanup can be reduced if not ignored.

In the paper, we also provide initial experimental results from an industrial case study, preliminarily evaluating the effects of the approach on the above stated benefits.

II. Theoretical Background

Software testing is traditionally performed by exercising test cases on the software under test. A test case consists of inputs (or stimuli) to be provided to the software under test, and the corresponding expected output (or response), to be compared to the actual response of the software under test in order to provide a test verdict (typically *pass* or *fail*).

For simple programs, single inputs or input vectors can be used to explore the behavioural space of the software. Each input vector, when executed on the software under test, produces an output that can be compared to the expected output for the software. For reactive systems, however, especially on higher levels of integration or system testing, test cases that use ordered sequences of inputs or stimuli in order to subject the system to specific scenarios are often used. This type of testing is known as scenario-based testing [6].

A. Automotive System Testing

Based on our experience, sceniario-based testing, using hardcoded 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 testing 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 [7]) automotive testing area (see e.g., [8] and [9]).

It should however be noted that scenario-based testing can be seen as a special case of the more general model-based testing approach [6]. 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. Declarative Testing

The approach proposed in this paper draws inspiration from previous work on *declarative testing* [10], [11]. The informal description of the difference between traditional testing and *declarative testing* is to switch test engineers' focus from *how* to perform tests to *what* to test [11]. Declarative testing focuses on describing the goals of a scenario instead of the steps needed to execute it.

Traditional testing techniques are not sufficient to cope with the non-determinism, various configurations and network topologies involved in distributed systems. Thus, a declarative testing approach was proposed using the Bloom language in [10]. In the approach, a framework is presented where the test engineer describes the input/output relation in a declarative test specification which an automated test system then utilizes to produce possible execution paths satisfying the specifications.

Another approach using declarative testing, initially conceived as a method for automated graphical user interface testing at Microsoft, is described in [11]. According to the authors, in the GUI test automation context a declarative testing approach can be used to reduce the number of test cases. Since the goal is expressed instead of the actions to reach the goal, duplications such as testing both a keyboard shortcut gesture and mouse gestures reaching the same end state can be avoided if only the end reaching the end state is of interest. Also, declarative testing increases maintainability of test code since declaration of interesting states are separated from the code describing the actions. Thus changing the software under test may sometimes only require changes to one entity instead of several.

III. Approach

This section provides a detailed description of the traditional approach of scenario-based HIL testing, as well as the proposed new approach using independent guarded assertions.

A. Traditional approach

In Figure 1, an example of a framework for integration tests is shown. This particular example is taken from the Swedish truck manufacturer Scania[1] but based on our experience with several other vehicular OEMs, we argue that it is sufficiently representative for test frameworks in general in this domain.

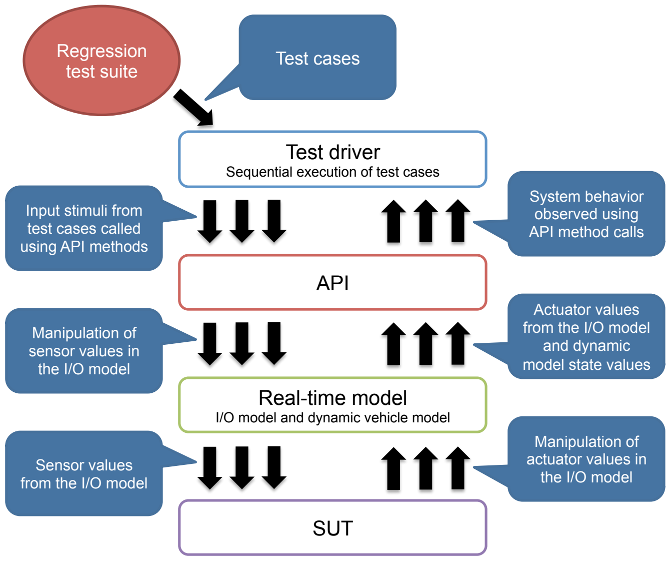
Regression test suite
Test cases
Test driver
Sequential execution of test cases
Input stimuli from test cases called using API methods
System behavior observed using API method calls
API
Manipulation of sensor values in the I/O model
Actuator values from the I/O model and dynamic model state values
Real-time model
I/O model and dynamic vehicle model
Sensor values from the I/O model
Manipulation of actuator values in the I/O model
SUT

Fig. 1. Integration Test Framework.

A test case for the framework represents some user function scenario that should be verified. A test case is typically structured into three phases according to the AAA principle of **A**rrange, **A**ct and **A**ssert. Sometimes a cleanup phase is also included. The general structure of a test case is depicted in Figure 2.

- The goal of the *Arrange* phase is to put the SUT in a certain state where the test scenario suitably can be tested (e.g., accelerating up to a certain speed or setting a certain temperature).
- In the *Act* phase, the steps in the scenario are executed.
- In the *Assert* phase the actual result of the previous steps is compared to some pre-specified expected result.
- In the *Cleanup* phase, the SUT is put in a state where execution of the next test case can commence without disturbances from the previous test case.

All phases can be constructed by means of sequences of input API method calls, sleeps and assertions comparing expected behavior with actual behavior. Based on the outcome of its assertions, a test case can, when exercised, render three different results: A *passed* test case is one with no violated assertions, a *failed* test case is one where the expected behavior

[1]www.scania.com

Arrange
0 . . . m of stimulus

Act_1
Stimuli 1
Stimuli 2
. . .
Stimuli n
Assert 1
Assert 2
. . .
Assert p
Act_2
. . .
Act_r

Cleanup
0 . . . s of stimulus

Fig. 2. An example of a structure of a test case.

does not match the behavior of the SUT (ideally indicating a fault in the SUT), and an *aborted* test case is a test case where some test case action cannot be performed (ideally indicating a fault in the test case or the test environment). A test suite is a set of test cases in a specific order. It is executed by a test driver, one test at a time.

This traditional approach has several limitations. Since test cases are executed sequentially, possible interactions between the behavior of several test case stimulus are not considered. For example, even though individual and separate test cases testing the reverse light and the hazard warning may exist, the combined effect of turning on the hazard warning while putting the gear in reverse would remain untested unless specific test cases were created testing this scenario.

The testing in each act depends on which stimuli is given. For example, the reverse light is tested by engaging reverse gear and checking the lamp, but it is not verified it is not lit for every other gear. Should such tests be performed, new acts must be developed, even though other scripts use forward gears for other testing purposes.

B. New Approach: Using Independent Guarded Assertions

In order to address the above limitations, a novel approach for automotive system HIL testing is presented. In our approach, instead of the traditional test case structure with arrange, act, assert and cleanup as described above, a test suite can be divided into two parts; 1) The assertions, guarded by conditions on the state of the SUT as described below, and 2) The stimuli that drives the SUT to satisfy the states needed to release the guards and trigger the assertions to evaluate.

1) Writing Test Cases as Independent Guarded Assertions: In the proposed approach, a test case is structured in a manner similar to the guarded command approach [12]. The *command* is the assertion that should be evaluated on the SUT and the *guard* is a condition on the state of the SUT defining when the assertion is valid to evaluate. This can be described more formally as $\{guard \Rightarrow assertion\}$, with the meaning that when the guard condition is satisfied the assertion is evaluated. For example, when gear is in reverse it is asserted that the reverse light is on. Consequently, when gear is *not* in reverse the assertion is not valid and should not be evaluated. In this example the guarded assertion can be formulated as {`gear=reverse` ⇒ `reverse_light=On`}. The test cases are formulated as *read-only* in the sense that the guards and the assertions of test cases are only allowed to read the state of the SUT but not to manipulate it in any way.

Using the proposed approach, a test case can be made to "wait" for a certain state of the SUT, and evaluate an assertion when that state is reached. For example, a test case "waiting" for the gear to be in reverse can - when that condition is satisfied - assert that the reverse light is on. Pseudo code of this example, using guarded assertions, is shown in Figure 3.

```
while(true){
  EventWait(gear == Reverse);
  Assert(Reverse_light == On);
}
```

Fig. 3. The guarded assertion structure as pseudo code.

The EventWait call makes the test case wait for the `gear` part of the state of the SUT. When the `gear` is `Reverse` the guard will release and the `Assert(Reverse_light == On)` will be evaluated and report failure if the reverse light is not on. The guarded assertion will execute within the outer infinite loop that continuously evaluates the assertion when `gear` is in `Reverse`.

Since the assertions do not affect the state of the SUT, it is possible to execute several independent guarded assertions in parallel without risk of them interfering with each other. Thus, an example test suite testing various lights and gears in a parallel fashion could be:

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.

Test 3 can be tested simultaneously with either test 1 or 2. Test 1 and 2 are mutually excluded.

2) Driving the SUT: In order to satisfy the guards of the assertions during testing, stimuli that change the state of the SUT accordingly must be provided. In the proposed approach, the script(s) providing the stimuli required to release the guards are executed in parallel with the assertion scripts, as shown at the top of Figure 4. Using the above example above with gears and lights, this provided stimuli should ideally ensure changing gears to reverse and neutral to release the guards of the assertions in the test cases.

Since the purpose of the stimuli is to put the SUT in a state that release the guards, they correspond to the stimuli used in

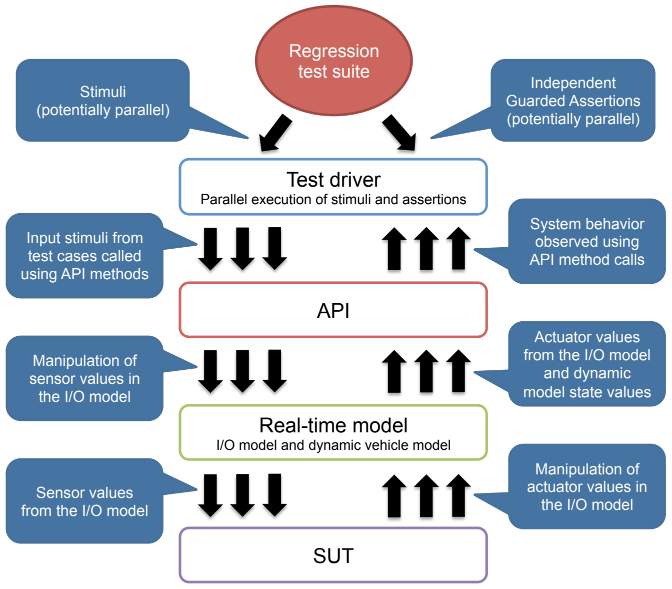
Regression test suite
Stimuli (potentially parallel)
Independent Guarded Assertions (potentially parallel)
Test driver
Parallel execution of stimuli and assertions
Input stimuli from test cases called using API methods
System behavior observed using API method calls
API
Manipulation of sensor values in the I/O model
Actuator values from the I/O model and dynamic model state values
Real-time model
I/O model and dynamic vehicle model
Sensor values from the I/O model
Manipulation of actuator values in the I/O model
SUT

Fig. 4. New Integration Test Framework.

a traditional test case's Arrange and Act phases. However, in a traditional approach the arrange phase is sometimes required because the test case should start executing from a known starting state of the SUT. For example, the first test case in a test suite execute from a "fresh" SUT state and may need to perform some state changes in its Arrange phase. Also, all test cases following a test case that restores the SUT to its original starting state in its cleanup phase may also require the Arrange phase. The cleanup phase in a traditional test case may be a requirement to avoid introducing dependencies between test cases where a certain test case cannot execute correctly unless some other test case has executed first. Consequently, several test cases may have similar arrange phases (e.g. speed up to a certain speed or turn ignition on) and also similar cleanup phases, e.g. put the gear in neutral or turn off engine. Such frequent (and often times unnecessary) sets and resets of system state may result in an inefficient test suite execution.

In the proposed approach, all independent guarded assertions may execute continously in parallel, the stimuli that cause SUT state changes are centralized, and thus the concept of restoring the state after an assert is no longer needed. Instead, the provision of the stimuli can be designed such that desired assert coverage of the test cases is acheived. This approach provides possibilities to have several parallel sources of stimuli aimed at certain goals, e.g. coverage goals where all asserts should be executed as fast as possible, to mimic a real-work usage of the SUT, or even to maximise fault detection (e.g., by evaluating the same assertions in as many different "valid" SUT states as possible). This is further elaborated in Section V-A future work.

IV. Experimental Results

This section describes the results of an experiment conducted to provide an initial evaluation of the benefits of the approach with respect to the objectives of increased real-world representativeness, reduced execution time, and increased defect detection. Section IV-A describes the experimental setup, and Section IV-B shows the results and the analysis.

A. Experimental Setup

We have executed an experiment focusing on executing multiple test scripts in parallel. Here we have taken 10 test scripts from Scania's full vehicle integration test suite and converted them into 68 independent guarded assertion scripts, and one stimuli script. The stimuli script, or the *course*, is a concatenation of all stimuli in the original scripts where timing and order is kept.

Scania's test automation framework is used. It is client/server based where each script constitutes a client, and access to the HIL is done via the server. The server supports multiple clients.

First, the server is started, then all assertion scripts are started, and when they are ready, the course script is started. The assertions are closed when the course script is finished.

B. Results and Analysis

Figure 5 shows a plot where the x axis shows time ($x = 0$ at start of course), and the y axis is allocated one integer value for each assertion, and $y = 1$ is allocated for the course. Each time the course script is providing stimuli to the system under test a dot is plotted on $y = 1$. A signal can, for instance, be pressing the hazard warning button. Each time an assertion has executed, a dot is plotted on the corresponding y value.

As can be seen from the figure, assertions are executed concurrently, and are performed more times than only once for a majority of the assertions.

Moreover, of all assertions that are performed at least once, they have done so at the time 374 seconds, which is 5 seconds faster compared to running the test scripts sequentially. Further, 9 assertions have been performed before their corresponding part of the course has started. These two results indicate that without optimizing the course against any objective, it is still possible to get benefits relating to time and concurrency.

We noticed that this initial transformation of the original scripts into guarded assertions gave three different results:

- the transformed script reaches the same verdict as the original script,
- the transformed script reaches at least one verdict that is different from the original script, and
- the transformed script is never executed (there are several such transformed scripts in Figure 5).

The last bullet can be due to mistakes in the transformation so the script is aborted by the runtime environment, but it can also be due to a too strict guard. The second bullet is most likely due to too insufficiently strict guards; they let tests against expected responses be executed even if the intended preconditions are not fully met. Based on these observations, we focused efforts on learning how guards can be constructed, described below.

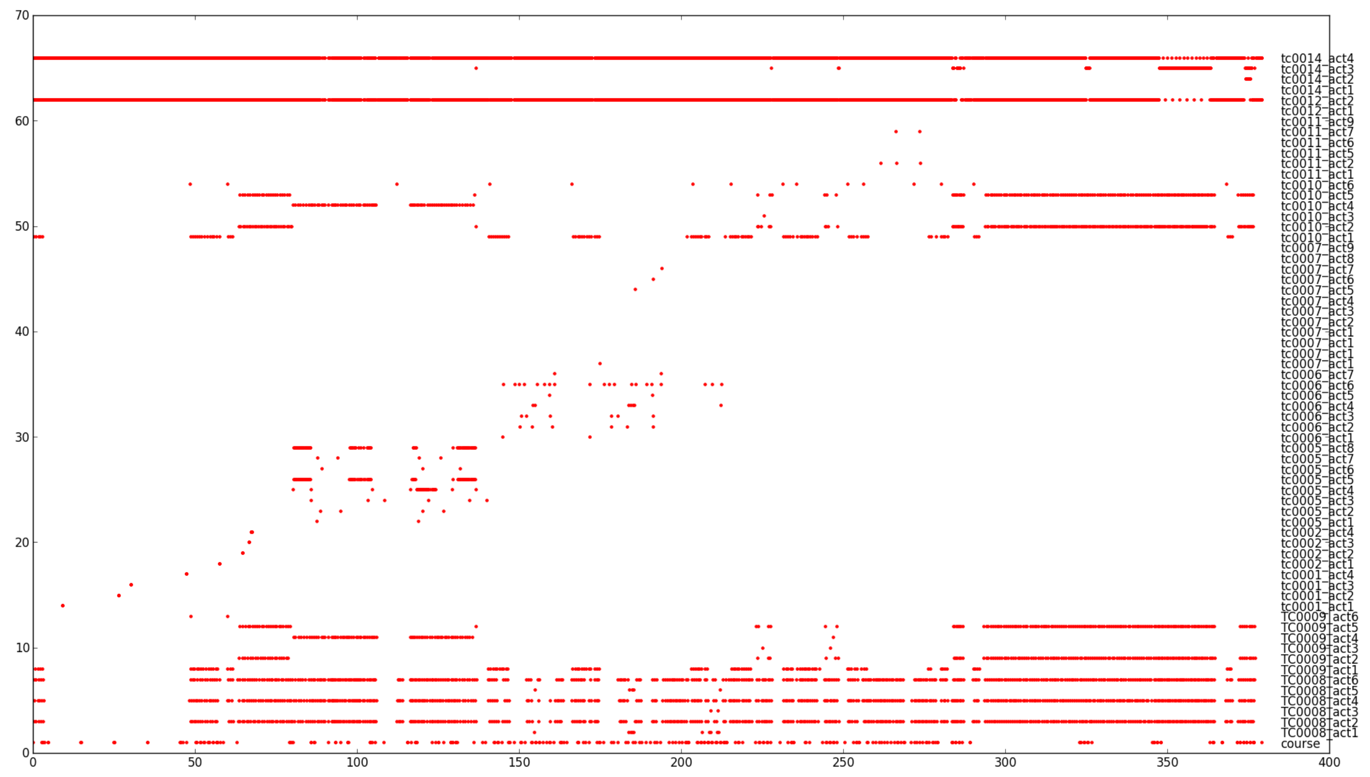

Fig. 5. Plots of stimuli in the course, and when assertion scripts perform assertions. x axis shows time in seconds, and y axis shows when assertions are performed or when the course performs a stimuli.

Test scripts 1 and 2 test different aspects of hazard warning. We saw that the verdicts of several guarded assertions differed from those of their sequential counterparts, e.g., assertion 2 of test script 1. We found that the assertion guard collects data for around 10 seconds, and then analyzes it. The hazard warning shall be activated during this period, but the course deactivated it too early for the assertion guard to be satisfied. This can be remedied by either correcting the course or constructing a guard that checks the functionality during data collection. Thus, there is a relationship between the course and the assertions that must be considered when constructing a course.

Test script 5 tests the worklight. By pressing a button, worklight lamps are enabled for easier maneuvering at low speeds. Furthermore, the button has two functions, pressing it always turns on/off the worklight, but if it is pressed for more than 3 seconds, then it activates/deactivates the automatic activation of the lamps if the reverse is engaged. Now we focus on assertion 4 and its transformation into a guarded command. The implementation of the guard of assertion 4 covers the used techniques of the other assertions of this test script.

Assertion 4 shall verify that if the worklight button has been pressed an even amount of times, i.e., the functionality is off, but the 3 second functionality can still be active, then when switching gear from reverse to neutral, the worklight lamps and the indication shall be off. The guard for this is given in Figure 6. Three concurrent threads of execution is needed to implement the guard. One checks the worklight switch, one determines when the gear is changed from reverse to neutral, and the third fuses the information into one decision: test or not to test. Thread 1 needs to consider that it takes time for the systems handle button presses.

V. Conclusions and Future Work

In this work, we set out to find a way to achieve the following objectives:

1) increased real-world representativeness,
2) shorter testing time, and,
3) improved defect detection.

We argue that being able to execute tests concurrently is one way to address all three bullets at the same time. However, executing tests containing stimuli concurrently has a risk of putting the SUT in a state that none of the executing scripts want it to be in. Our way of resolving this problem is to utilize independent guarded assertions.

In our experimental evaluation, 10 test scripts, divided into 68 acts, were transformed into 68 independent guarded assertions and then executed concurrently with a script representing the course. The conclusions are as follows:

- Some assertions are triggered only once, but many are triggered multiple times, where each additional time is a new possibility to find a fault.
- The guards must be complete, otherwise assertions are triggered at the wrong occasion, or not triggered at all.
- There is an intricate relationship between the executability of a script's assertions and the stimuli in the course.

Thread 1
```
while(true) {
 EventWait(WorklightSwitch != Off)
 EventWait(WorklightSwitch != On)
 if worklight {
  worklight=false;
  negative_edge = true;
 }
 else {
  worklight = true; negative_edge = false;
 }
 counter++;
}
```
Thread 2
```
while(true) {
 EventWait(Gearbox != GearboxInReverse)
 status = false
 EventWait(Gearbox != GearboxNotInReverse)
 status = true
}
```
Act Thread
```
while(true) {
 Wait for Gearbox != GearboxNotInReverse
      and Thread 2 status is true
      and Thread 1 counter is even
      and (Thread 1 negative_edge is false
      or  Thread 1 negative_edge is true
                for at least 10 ms)
 Perform tests
}
```

Fig. 6. Implementation of the guard of assertion 4 in test script 5.

- The testing time can be decreased even though the same stimuli as in sequential testing is given.
- Multiple assertions are triggered simultaneously even without optimizing the course for this.

The above results indicate that independent guarded assertions is one way to change how one reasons about testing, yet getting benefits in terms of representativeness, testing time, and default detection.

A. Future Work

As for future work, a larger, more rigorous empirical evaluation of gains acheived in terms of efficiency and effectiveness is called for. Related, another important direction for future work concerns construction of stimuli sequences. In our initial approach, stimuli are executed in the way they were defined by existing legacy test scripts, including setting up preconditions and returning the SUT to some safe state after test evaluation. Stimuli sequences could however be constructed to meet one or several optimization criteria, such as fault detection, assertion coverage, or execution time minimization.

In terms of practically integrating the proposed approach into everyday testing practice, evaluating and (if needed) improving scalability is a concern. For example, in our current implementation, each independent guarded assertion is continuously polling a signal database in order to detect appropriate conditions for test evaluation. This could be replaced by a hierarchical approach, where frequently used signals are examined by a common poll mechanism, which in its turn trigger appropriate test evaluations. We are also starting to look at alternatives where test logs are processed and evaluated post-mortem.

Acknowledgments

This work has been partially funded by VINNOVA, Sweden's innovation agency, within the FFI program.

References

[1] M. Young and M. Pezze, *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005.
[2] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge University Press, 2008.
[3] 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
[4] 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
[5] 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
[6] 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.
[7] 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
[8] 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.
[9] 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.
[10] 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.
[11] 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.
[12] J. L. Wagener, "Guarded command," in *Encyclopedia of Computer Science*. Chichester, UK: John Wiley and Sons Ltd., pp. 761–762. [Online]. Available: http://dl.acm.org/citation.cfm?id=1074100.1074433
[13] D. Sundmark and A. Kobetski, "Parallelization of integration tests, prestudy report," SICS, Tech. Rep., May 2013.