2016-01-0032
Published 04/05/2016
Copyright © 2016 SAE International
doi:10.4271/2016-01-0032
saepcelec.saejournals.org

Incorporating ISO 26262 Concepts in an Automated Testing Toolchain Using Simulink Design Verifier™

Siddartha Khastgir, Gunwant Dhadyalla, and Paul Jennings
WMG, University of Warwick

ABSTRACT

The introduction of ISO 26262 concepts has brought important changes in the software development process for automotive software. While making the process more robust by introducing various additional methods of verification and validation, there has been a substantial increase in the development time. Thus, test automation and front loading approaches have become important to meet product timelines and quality. This paper proposes automated testing methods using formal analysis tools like Simulink Design Verifier™ (SLDV) for boundary value testing and interface testing to address the demands of ISO 26262 concepts at unit and component level. In addition, the method of automated boundary value testing proposed differs from the traditional methods and the authors offer an argument as to why the traditional boundary value testing is not required at unit (function) level. There are two aspects of the proposed method: automated test case generation and automated test case execution. The paper discusses the benefits of automatic test case execution when combined with automatic test case generation. Traditional test automation implements the former and has limited advantages. One of the challenges with traditional application of the formal analysis tool is the time taken by the tool to reach to a conclusive decision for the triggered activity, i.e., the execution time of the tool. This shortcoming is overcome by an automated setup where the test framework is triggered during out of office hours, which saves developer's work time. As a work product of the automated test execution, the developers receive test documentation which provides them with an overview of the results and specific test vectors for further analysis.

CITATION: Khastgir, S., Dhadyalla, G., and Jennings, P., "Incorporating ISO 26262 Concepts in an Automated Testing Toolchain Using Simulink Design Verifier™," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 9(1):2016, doi:10.4271/2016-01-0032.

INTRODUCTION

Over the past two decades, the software content in road vehicles has increased exponentially. It is said that cars have over 100 million lines of code compared to a jet fighter which has just over 1.7 million lines of code [1]. Most of the upcoming innovations in the automotive industry will also be due to software. With the increase in driver assistance and autonomous technologies, increased importance will be placed on rigorous Verification and Validation (V&V) processes which need to be established. The ISO 26262 standard [2], attempts to add clarity for V&V practices that should be followed for automotive software development to meet the functional safety goals. While it is imperative to incorporate the development practices as detailed by ISO 26262 in the software development process, it requires additional time and adds cost to the development process.

Moreover, the industry is always hard pressed for time and availability of resources for the introduction of new products. Therefore, test automation within existing agile processes needs to be developed and adopted to incorporate the concepts of ISO 26262. Test automation has two components: automated test case generation and automated test execution. The former can be based on various mathematical theories, like formal verification, random testing, equivalence testing etc. The latter is influenced by the existing software development processes in an organization and its incorporation within the development workflow. Automated execution of tests has increasingly been adopted by the industry, however, automated test case generation has a very low uptake [3]. It is worthwhile to note that despite the presence of industrial formal analysis tools like Simulink Design Verifier™ [4], BTC Embedded tester [5] etc., only 8.6% of the respondents in a survey conducted by [3] responded positively to using automated test case generation. Further, the findings in [3] suggest that this is because of the limitations of the tools being used for automated test case generation. Another contributing factor for the low uptake is the inability of the current processes to exploit the complete tool capacity [6]. This is borne out by the authors' own experiences of practical testing of systems in industry.

There is even less evidence of automated test case generation related to non-functional system attributes such as Boundary Value Analysis and Testing specifically to satisfy ISO 26262. [7][8][9][10] focus on high level test planning, test frameworks, testing workflows and the role of Hardware-in-the-loop testing respectively. Where model based approaches are suggested, they relate to the use of model based development with fault injection using mutation testing at a function level rather than system level [11][12], or the use of automated testing for ISO 26262 requirements-based functional testing of model-based embedded software [13]. Other publications focus on

Documents
Document Sharing Hub
Full document details: SAE 2016-01-0032

static testing for structural analysis rather than requirement testing, evaluating performance attributes such as timing, stack overflows etc. [14]; monitoring proof obligations by the use of an on-chip, real-time runtime verification monitor [15].

TYPES OF TESTING

ISO 26262 part 6 defines different methods for software unit testing. These include Requirements-based test, Interface test, Fault injection test, Resource usage test, Back-to-back comparison test between model and code (Figure 1). For each of these methods, the standard suggests different methods for deriving test cases (Figure 2). These include, analysis of requirements, generation and analysis of equivalence classes, analysis of boundary values and error guessing. It should be noted that interface testing is highly recommended for all ASILs (Automotive Safety Integrity Levels) of software.

	Methods	ASIL A	B	C	D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	+	++
1d	Resource usage test[c]	+	+	+	++
1e	Back-to-back comparison test between model and code, if applicable[d]	+	+	++	++

[a] The software requirements at the unit level are the basis for this requirements-based test.

[b] This includes injection of arbitrary faults (e.g. by corrupting values of variables, by introducing code mutations, or by corrupting values of CPU registers).

[c] Some aspects of the resource usage test can only be evaluated properly when the software unit tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

[d] This method requires a model that can simulate the functionality of the software units. Here, the model and code are stimulated in the same way and results compared with each other.

Figure 1. Methods of software unit testing as defined by ISO 26262 part 6

	Methods	ASIL A	B	C	D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes[a]	+	++	++	++
1c	Analysis of boundary values[b]	+	++	++	++
1d	Error guessing[c]	+	+	+	+

[a] Equivalence classes can be identified based on the division of inputs and outputs, such that a representative test value can be selected for each class.

[b] This method applies to interfaces, values approaching and crossing the boundaries and out of range values.

[c] Error guessing tests can be based on data collected through a "lessons learned" process and expert judgment.

Figure 2. Methods for deriving test cases for software unit testing as defined by ISO 26262 part 6

Boundary Value Testing

Boundary Value Testing is a type of stress testing, where stress testing is defined as "testing conducted to evaluate a system or component at or beyond the limits of its specified requirements" [16] and a boundary value is defined as "a data value that corresponds to a minimum or maximum input, internal, or output value specified for a system or component" [16].

For example, a unit with an integer input range between 1 - 99, would need to be tested with values that lie outside the specified range (in this case with e.g. 0 on the lower side and 100 on the higher side). Traditionally, each interface present in a software needs to be tested for boundary values. In Figure 3 that would mean interfaces between system Ext 1 and system A, system A and system Ext 2, and the interfaces within system A (each smaller block being a unit e.g. *Fa*, *Fb* etc.), for the software of system A to be compliant with ISO 26262. Depending on the system size, this can be a time and cost intensive task.

Traditional Boundary Value Testing

Consider a function *Fb* that has 6 inputs and 6 outputs. Each input should be tested for both upper limit and lower limit (crossing boundary values) and the combinations with the other inputs. This would mean in order to perform complete boundary value test, one needs to generate 3^6 test cases (729 test cases) manually following the equivalence class method. This is not only time consuming but also not pragmatic. The number of test cases for a large number of functions can easily reach many tens of thousands.

The proposed method of interface testing, serves as a replacement for traditional boundary value testing (as specified by ISO 26262) for system's internal functions or units, the units inside system A, Figure 3. It should be noted that the proposed method doesn't hold true for system-system interfaces (e.g. interface between system Ext 1 and system A in Figure 3).

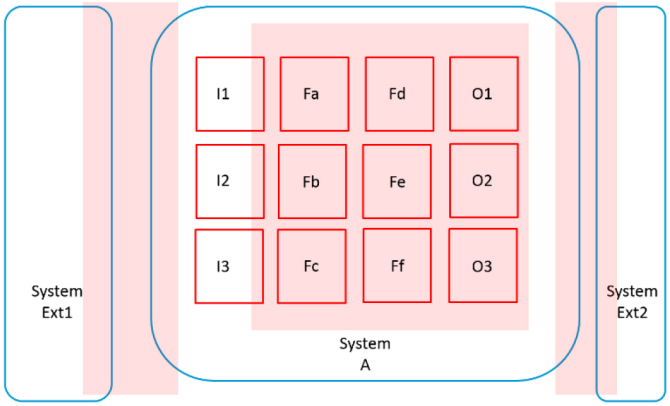

Figure 3. Schematic depiction of system and system's unit's internal boundaries

HYPOTHESIS

The hypothesis offered is that if it can be proven that the outputs of internal units or functions can "never" lie outside the specified ranges (for any legitimate value of inputs of the unit), then there is no need to test boundary values for the next unit given that the output signal of every unit will be an input signal for another unit. Taking the example of Figure 3, if it can be proved that for the units I1, I2 and I3, the outputs can never lie outside the specified range as per requirements, for all values of inputs (lying within the specified range), boundary value testing for Fa, Fb, Fc need not be done as all inputs (outputs of I1, I2 and I3) will always lie within specified limits. However, once again for Fa, Fb and Fc, the interfaces need to be tested to prove that outputs can never cross specified ranges for any legitimate input values. This is a recursive process and is valid as testing moves downstream within system A.

METHODOLOGY

In this paper, the authors will discuss using automated test case generation for the analysis of boundary values and for interface testing. The authors discuss the use of Matlab Simulink® Simulink Design Verifier™ (SLDV) [4] toolbox as an example tool for the illustration of the hypothesis developed. This tool has been chosen as it is a TÜV SÜD certified tool for use in ISO 26262 compliant

Document Sharing Hub
Full document details: SAE 2016-01-0032

development process [17]. Additionally, since Matlab Simulink® is a de-facto industry standard modelling and simulation tool for automotive software development, Simulink Design Verifier was chosen as it can work on MATLAB Simulink® models. SLDV uses formal methods to generate test cases (time series vectors) for design errors, property proving, coverage maximization [18]. While the know-how about formal tools like SLDV has been well documented for requirement based testing [19] [20] [21], (where requirements are modelled as properties using SLDV library blocks), its use in software development process and for analysis of boundaries and interfaces has not been reported. This has been backed by the results of the survey in [3] where only 5.21% of responses accepted the use of formal methods in their mass productions projects, even though the existence of formal methods and tools was known to them.

Due to the potentially large number of test cases that can be generated, any formal analysis tool like SLDV can have long execution times, sometimes many hours. In order to overcome this, the authors propose an automated testing framework which is triggered during out-of-office hours to reduce the time spent by system developers waiting for test results to be generated.

Automatic Test Case Generation using Simulink Design Verifier

As a pre-requisite to perform the proposed interface testing, a Matlab Simulink® model of the software unit and its requirement specification for the interfaces are needed. The following steps need to be followed for the proposed process (flowchart explained in Figure 4):

Step 1. Identify if the software unit has a system-system interface(s) or a system-internal interface(s).

Step 2. For a system-system interface, perform traditional boundary testing. If the outputs of the interface model lie within the specified range (as per requirements) for over and under boundary limit values of inputs, then proceed to step 3. Else, re-visit requirement specifications to check for anomaly.

Step 3. For a unit with system-internal interfaces only (e.g. Fa, Fb etc. in Figure 3), identify the inputs and their value ranges as per requirement specification.

Step 4. Constrain input ranges for the input signals to the specified values. (For SLDV, the Assumption block at the root level may be used).

Step 5. Define the proof objective for the formal analysis tool as the specified range of each of the output signal. (For SLDV, specified range for the output signal can be modelled using Proof block from SLDV library).

Step 6. Trigger formal analysis tool to find violation to the defined property. (For SLDV, SLDV should be triggered in “Find Violation” mode to falsify the defined property (generate a test case (counterexample) for which the outputs of the unit lie outside the specified range)).

Step 7. Analyze results/report generated by the formal analysis tool. If no test case exists (counter-example) where property is falsified, boundary value testing doesn’t need to be performed for the unit. (Steps 3-6 can be completely automated).

Step 8. In case of a violation of property, re-visit requirement specification to understand the system behaviour. (In case of SLDV, a counter-example test case causing the violation will be generated).

Step 9. If for all internal software units, the formal analysis tool can’t generate a test case for which the outputs violate the specified limits, then limit traditional boundary value testing to system level interfaces only (i.e., for the interface between system Ext 1 and system A and between system A and system Ext 2 in Figure 3)

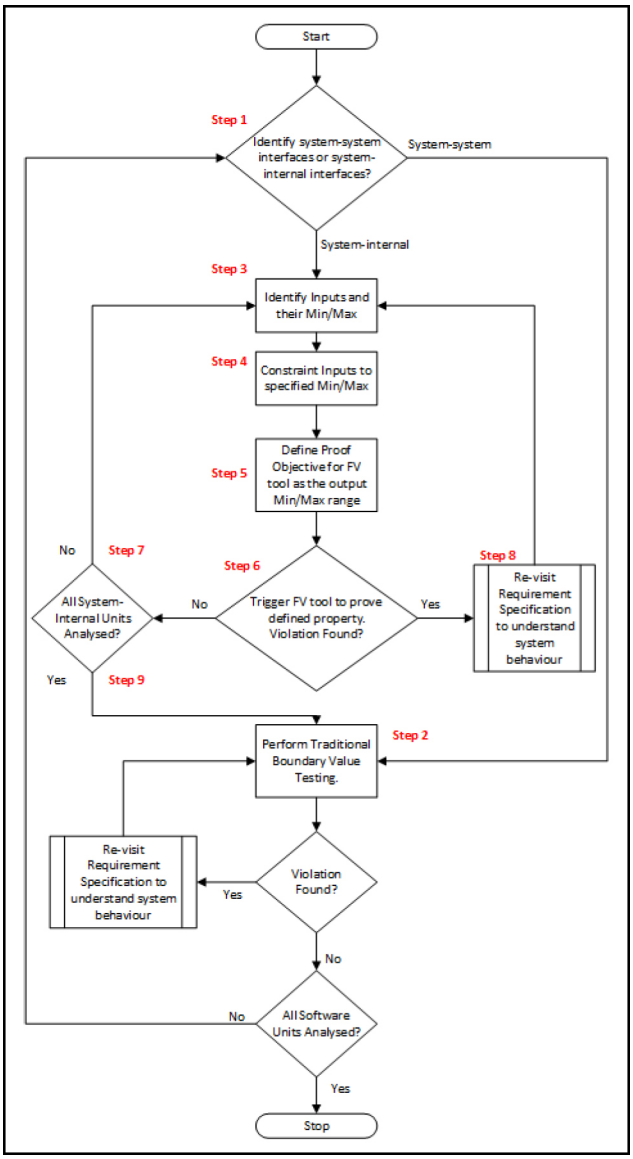

Figure 4. Flow chart of proposed work process

In order to constrain the state space for formal verification by SLDV (to prevent the explosion of possible combinations), the input signals of the unit are constrained to the ranges as per the requirement specification, i.e. it is assumed that the inputs will always be within the specified range. While using SLDV, this is achieved by incorporating the “Assumption” block from the SLDV library. The

AVA Documents Document Sharing Hub

Full document details: SAE 2016-01-0032