ABSTRACT:

Model based software development is emerging in automotive embedded industry as more software components in the automotive systems are developed using Models. As the embedded systems become more and more complex, it increases the need of having a more robust method that aids fast and accurate system-level estimation. When implementing production software through AUTOCODE PROCESS for fixed-point electronic control units (ECUs) it is important not only to consider the code optimization but also the compliance of the code with the model. Model consistency checks are done to ensure that the generated TARGETLINK model replicates the functionality of the MATLAB-SIMULINK model and accounts for the numerical quantization results from the fixed point implementation. Autocode generation invoked with options that maximize code efficiency for fixed-point processors also impose an exhaustive review and testing conditions. Also it becomes more critical that the interfaces between the modules are used with the right attributes consistently. This paper on the autocoding verification process proposes tools that help the peer review process, the testing process and the integration of the autocode into the underling layers by automating the necessary checks considering the above fixed point constraints. These tools are developed based on the analysis of the repetitive occurrences of similar kind of findings over the years of autocoding process.

INTRODUCTION:

The growing complexity of the automotive systems, short production release schedules and the safety critical algorithms of the software make the task of verification challenging. Review and testing play a major role in the process of verification, traditionally manual review and testing incur a considerable effort, time and cost. Further when bugs are detected, the cost of rework and additional testing is high, impacting the quality of the product and the lead time. The increased complexity of today's software-intensive systems requires a detailed and fool proof verification process that can be achieved by automation. This helps in detecting the errors at the earlier stages of the development and also eases the process of verification.

Though there is detailed verification strategies followed in the verification procedure, they do not address the costly issue of catching bugs without investing significant amount of effort. The peer review tool is not only easy to understand but also powerful enough to run checks on the Target Link model. This will ease the peer review process as it addresses the basic and common noncompliance that occurs during the auto code generation. The test automation tools may not replace human intelligence in testing, but without them testing complex systems at a reasonable cost will never be possible. These testing tools automate the verification of the autocode against the spec model that is used for the generation of the autocode. The integration tool runs necessary checks to ensure proper integration of the autocode into the software.

The shortcomings in the manual verification process when used to verify software-intensive, safety critical systems can be addressed in the tool with measurably higher quality in lesser time ensuring efficient verification process.
BOTTLENECKS IN MANUAL VERIFICATION:

When reviewing the Targetlink model and the generated code, certain checks based on standards and the lessons learnt over years for different products have to be verified. With the manual review method there are possibilities of missing out some of these checks and some checks are proven to be tedious, there by making the manual review less reliable.

The manual testing process becomes difficult during the root cause analysis of the failures to trace the exact instant of the failure. It involves greater effort particularly in complex math-intensive algorithms. Hence there is a need for a tool which can show the exact time instant of the failure, the combinations of the inputs that caused failures and inter-dependencies of the failures.

APPROACHES TO VERIFICATION AUTOMATION:

The verification process can be carried out at three stages:

1. Verification/Review of the Targetlink model
2. Testing of the generated code
3. Verification at the integrated software level for the interfaces.

VERIFICATION OF MODEL COMPLIANCE FOR CODE GENERATION:

The automotive model-based developments usually consist of complex functionality and the systems interact with physical components that have a complex behavior which depends on many variables. The interface among the modules in the model and with other physical components is critical and the errors at these levels of interface are undesirable, posing a threat to the functionality of the system. Though review of the core functionality cannot be automated but there are some repetitive checks which follow a fixed pattern that can be automated. Such checks are largely error-prone with the manual review when done under extreme schedule constraints. Hence there is need of a tool to automate such checks and to make the verification process robust and faster.

The tool developed is based on the checks that are recommended by various autocode standards, best practices, lessons learned from the functional defects due to incorrect property setting in the model and some limitations in the code generation tool. The defects can be due to various root causes such as the stateflow parameters not initialized from workspace, calibration constants with inappropriate data type or class, unintentional typecasting leading to overflows or underflows etc. Such defects are difficult to be caught in the manual review always unless a detailed review is performed that consumes considerable effort.

The tool also ensures the consistency of the model and refines the errors that can be caught at the earlier stages and also in lesser time without manual intervention. This improves and assures higher quality as it eliminates the defects of the TargetLink model before the coding starts, aiding significant cost saving. The tool automatically ensures industry guidelines, such as MISRA and Delphi autocoding standards, are practiced consistently in the organization. Some sample checks that tool supports are discussed below:

Checks on global interfaces:
- The Targetlink blocks with the variable class field global should have a reference to a data dictionary variable which ensures that attributes are reused from a central repository containing data for all global variables and there by ensures right data type and scaling are used by all modules for the same variable.

Checks on Lookup tables:
- The look up table method extrapolation is generally not recommended in embedded system as this leads to undesired outputs in some cases due to possible overflow. Catching this by manual review for complex systems having innumerable lookup tables is tedious and error prone.
For a lookup table, the tool compares the attributes of the output signal with that of the output axis, and the attributes of the input signal with that of the input axis.

Checks on calibration attribute:

- Check for the datatype compatibility of the calibration constants in the central calibration data dictionary with the datatype specified for the calibrations in the model. There are additional checks to determine if the datatype accommodates the range and resolution of the calibration constants.
- The consistency of the calibration constants used across other modules is also checked.

The report from the tool consolidates the compliance of the TargetLink model to the checks and sample report is shown below.

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>Peer Review Tool Report</td>
<td>Non Compliance</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model</td>
<td>Pass</td>
<td>FAIL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check 2: Constants which are not initialized from the workspace use proper initializations for constants</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check 3: Functions which have to be made reusable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check 4: Top level interfaces whose properties are not selected from the LDU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check 5: Default module ID used, specify a particular module name to prevent issues during building</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check 6: No links at the paths are broken or disabled</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check 7: Datatype of the input to the look up table does not match with x axis calibration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check 8: Datatype of the Output of the look up table does not match with y axis calibration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check 9: Equivalent with not supported by TL to generate code; change the type in the Look Up Table</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Report generated by the Peer review tool

It is critical to build these tools with a good architecture and design for easier understanding and maintenance. This is important as the tool needs to be easily updated to implement a new standard or custom check based on the need of the project or a new lesson learnt. Tool implemented by Delphi is neatly architected so that updating the tool is easy and a new check can be added with simple modifications.

AUTOMATION IN TESTING AUTOCODE:

The Auto code generated has to be verified for adherence to its functionality. This is generally done by running the functional test vectors on the model and code and comparing the outputs. The functional test vectors can be simulated data or real time logged data from vehicle or test bench. Additional test vectors can be created manually to test for specific functional scenarios, for path coverage including MC/DC coverage and for range coverage.
Though a standard test vector template is suggested, depending on the tools used the format of the test vectors can be different. The test data format logged using the instrumentation tools connected to a test bench or data from a rapid prototyping system or path coverage test vectors generated using tools like Reactis could be different. A standard format for all these test vectors is suggested as it will be easy to understand, maintain and also enables using a common tool for testing and generating a report. There is a need to automatically convert the test vectors from one format to another based on the approaches used in a specific program. Reactis uses matrix format with set of input values, set of expected output values with periodic time stamp. It is easy to understand and to add test vectors and hence is one of the suggested standard formats for test vectors. Custom tools were created to convert the data from one format to another. Add on tools were also created to update the expected output from the Simulink model in case the same input scenarios have to be played back for different variants of the model or for different releases where the model might be changed but the test data log may not be captured for all the scenarios required. Alternatively Reactis can also be used to update the outputs from the model for a given scenario. Reactis can also provide the coverage metrics which can be a yardstick to measure the effectiveness test vectors.

Once the test vectors with expected output are ready, it has to be run on the generated code to ensure code adheres to the model. To run the test vectors manually and compare the results of the model against the code with an allowable thresholds for each periodic cycle is tedious and error prone especially when the scenarios are running for long duration. Also it is important to capture the report of the different scenarios tested and these reports can be used for certifications or as part of the customer acceptance. A custom tool was developed to run the field test cases on the Autocode in Software in Loop (SIL) or Processor in Loop (PIL) mode. The key advantage of this tool is that it logs the output values of both the spec model (expected output) and the code (actual output) with respect to each time step as shown in the snapshot below. Tool allows automated running of many vectors together in batches and it also displays a summary report of all the different scenarios. Subsequently only the failed test vector can be analyzed and mismatches, if any will be highlighted by the tool. This will be easy for debugging: to track the vectors failing and also indicates the time step when a particular test vector fails, this will be useful to track the inception of the problem and hence debugging.
VERIFICATION OF THE INTEGRATION OF THE AUTO CODE INTO THE UNDERLYING SOFTWARE LAYERS:

Layered architecture is generally used in embedded systems. Application layer consists of modules implementing the control algorithm. These are less hardware dependent and hence reusable. Whereas lower layers are more hardware dependent and include communication, diagnostics, memory, microprocessor specific startup code etc. The auto generated code from the Targetlink models (Application/Algorithm code) has to be integrated into the lower layers of software which are usually hand written. To achieve proper functioning of the entire system it is critical to ensure the interfaces among the auto coded modules, and interfaces between the modules in application layer and lower layers. Ensuring the interfaces between modules in application layer is achieved using central variable depository. In Targetlink models, these variables are referenced from such a common repository which is the Targetlink Data dictionary (TL DD) thereby ensuring the same attributes are used in all auto code modules.

However in cases where the interface variable is a hand coded variable, verifying such a variable with interface variables created in TL DD is very tedious and error prone especially with complex and large auto code modules that may have large number of interfaces. Such verification is absolutely vital especially in case of fixed point auto coding. For example a mismatch in the scaling information between the hand code variable and its corresponding entry in the TL DD will lead to generating auto code with...
wrong scaling operations performed on the variable. Such an error can be undetected by the compiler as the base data type used at both the places may still be the same. Interface datatype differences maybe introduced during the development phases where hand code and Autocode are being developed / updated in parallel. Hence interface verification needs to be done for every release towards the end but with less effort and time. A tool has been developed to automate the verification of the integrity between low layer software and autocoded software. Such tool parses the information from different sources like Targetlink Data dictionary, Autocode source files and Hand code source files. The output from the tool is a report highlighting any mismatches as shown below

![Delphi Interface Tool Report]

**CONCLUSION**

Automation in verification plays a vital role, considering the decrease in effort and increase in quality. Though manual intervention during verification cannot be replaced, it is always possible to develop tools that aid us during verification. The approach is to strike a right balance between automation and manual checks for better verification. In our process we have followed automation wherever a well defined pattern is identified and the tools are incorporated into the process to increase robustness. Detecting interface bugs prior to a product release is vital as the functionality changes drastically due to mismatches in interfaces. With the introduction of Peer review and Interface verification tools the defects that are caught during later stages have decreased drastically as most of them are caught at the auto code generation and integration phase itself. Many teams have successfully adapted these Automation tools, proving very effective especially during short production releases.

**ACKNOWLEDGMENTS**

The authors would like to acknowledge and thank the Modeling and autocode teams in Delphi Technical Centers in Bangalore, India; Kokomo, USA and Shanghai, China for all the contributions and guidance in development of the tools and process described in the paper.

**REFERENCES:**

2. [http://www.dSPACE.com](http://www.dSPACE.com)
4. Modeling style guidelines for the application of Targetlink in the context of Automatic code generation – MISRA.

7. “Defect Identification with Model-Based Test Automation”- Mark Blackburn, Aaron Nauman, Bob Busser, Bryan Stensvad


CONTACT:

nishanthi.duraisamy@delphi.com
sebastian.mathew@delphi.com
m.charantimath@delphi.com