Verifying and Validating Automatically Generated Code

Andrew Burnard, Land Rover

Introduction

The embedded software, developed in the automotive industry, has become increasingly more complex whilst the time to develop the systems has been reducing. This has required the process used to develop electronic control systems to make more use of computer simulation. The simulation allows earlier validation of the system being developed before representative vehicles are available. But there is still a significant effort required to develop and validate the code. With the advent of production code generation technology it is now possible to 'develop' the code easily. But how much can the automatic code generation technology be trusted? This paper covers issues raised in validating the code produced by the use of an automatic code generator.

Previously, when using manual generation of code, verification and validation is required to assure the 'quality' of the code. Techniques that have been used to verify and validate manual code include: reviewing, module testing, integration testing, system testing, and static analysis. These techniques help to find errors that engineers make when developing code. When using an automatic code generator, there should be no random errors that are typical of manual code generation, but any errors should be more systematic because the tool should always to the same thing. Production code generation for embedded automotive controllers is a relatively new technology. The first production code generators were available around the turn of the century, so it is reasonable to be cautious with the code produced by a tool.

The V-model is a useful representation to illustrate what is being covered by the verification and validation (V&V) of automatically generated code. The left side of the V describes the design process for creating the code, starting from the high-level system requirements and becoming more detailed at every step until the code is created. The right side of the V describes the verification and validation steps applied to each of the development steps.

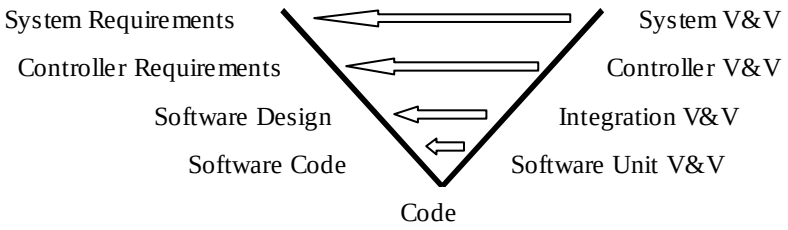

Figure 1: A V-model for manual code generation, without model development

With manual code generation, and no model development each of the verification and validation activities requires the previous step to be completed. This results in a significant delay in validating the system.

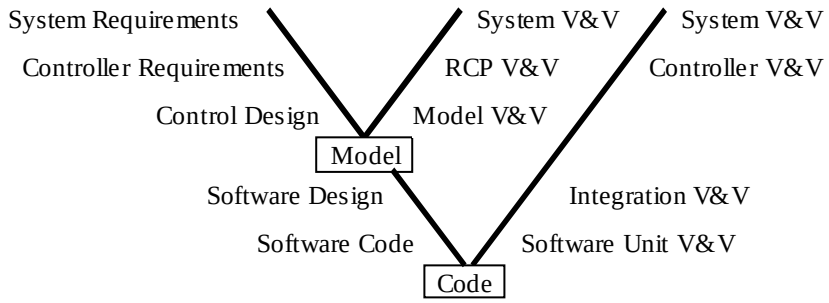

Figure 2: A V-model for manual code generation, with executable model development

When an executable model development step is introduced it is possible to perform some verification and validation of the system requirements using rapid controller prototyping (RCP) technology, but the final system still requires a code design and development stage, together with the required and time consuming verification and validation of the controller.

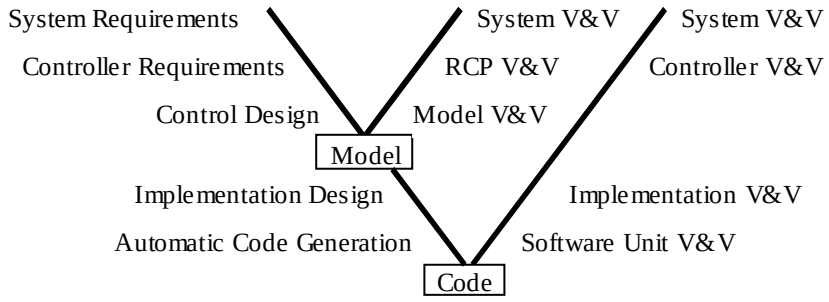

Figure 3: A V-model for automatic code generation

When an automatic code generator is used, the software design is replaced with a much simpler step of providing the code generator with information about how to generate the code (implementation design) and the automatic code generation step is 'simply' pressing a button. But good design practices need to be applied to the development of the model; this is checked for during model V&V and may require iteration in the design of the model. The software unit V&V is required to determine that the code generator has generated correct code and the implementation V&V is required to determine that the defined implementation is an accurate representation of the model. The controller and system V&V is still required to determine that the controller hardware and software will control the system as required, but this should be more straightforward than with manually generated code because the automatically generated code should behave more like the RCP system than manually generated code.

It is generally accepted within the safety critical community that if the results of code generated by tools is treated in exactly the same way as code produced manually, with regard to V &V techniques, then likelihood of errors being present in the final system is not increased but may actually decrease due to the absence of the coding errors typically made as a result of human error. While this is an advantage of using code generation tools a greater advantage would be achieved if some of the V&V effort could be reduced, in particular regarding the review of the source code. The purpose of this paper is to investigate methods of performing software unit V&V and implementation V&V for automatically generated code and argues that it is reasonable for a complete review of the source not to be performed. It is assumed that the controller and system V&V are preformed separately.

Techniques for verifying and validating automatically generated code

There are 2 different approaches to verifying and validating automatically generated code. The first involves validating that the code generator will be correct for any model being coded, and the second involves verifying that the code produced by the code generator is correct for a particular model. With manually generated code emphasis is placed on validating the code, because it is not possible to ensure that a programmer does not make mistakes; although mistakes can be mitigated against by only employing staff with the appropriate education, experience and training and also by the use of tools that have been developed to help a programmer make less mistakes, e.g.: the use of a high level language, a coding sub-set and style guidelines.

Validating a code generator

Validation of the code generator is primarily the tool vendor's responsibility, but it is unlikely that users of a production code generator will trust the code generator until they have had significant experience in the tool.

When a user has validated a code generator, the validated version of the tool and any configuration options applied to the validated tool will need to be used throughout the code generation process. This implies that if the configuration options are changed, or the code generator version changes, or a patch is applied to the code generator, the validation process will need to be repeated!
There are a number of techniques that can be used to validate the correctness of a code generator. The techniques concerned all increase the users confidence in the correctness of the code generator, but it is unlikely that any of the techniques will produce a perfect tool, due to the complexity of a code generator.

Tool Development Process
The process used to develop the code generator will improve the quality of a code generator. The tool developer should have a credible process for developing the tool. There are various means of assuring a user that a credible process has been followed:

- Presentation of the process. This is of some use, but without proof of the process being followed the confidence in the tool may be minimal.
- Performing audits on the process. These audits should be independent of the tool development process
- Certified process, e.g. SPICE, TickIT, CMM

If the tool supplier provides proof of a credible process being followed then the user of the code generator will be more confident in the tool concerned.

Testing the code generator
The user of a code generation tool can gain confidence in the tool by running their own tests on the tool. These tests supplement the tool suppler tests and would be more tailored to the user's use of the code generation tool, rather than more comprehensive testing which is what the tool supplier would need to achieve. Rigorous testing of a code generator is not always possible, because consider for example a simple addition (a=b+c) there are more than 2000 ways of implementing this depending simply on the data types and whether data limiting is enabled. Any testing a user applies to a code generator tool will be limited by the resource available to the user.

The automatic code generation tool could be certified to a recognised standard. This would probably increase the cost of the code generator by a factor that would price it out of the automotive industry but may be required for other industries [1].

The only rigorous method of validating a code generator would be to use formal proofs. This involves mathematically proving that the code generation transformation process is correct. Unfortunately the modelling notation itself would need to be defined formally first and, furthermore, the amount of effort required to perform a formal proof on a complex code generator would be prohibitive for the user but may be considered by the code generator supplier.

Experience
Users of a code generator gain confidence (or lack in confidence) through the experience that they have with the code generator. If the code generator is a quality product, the user will be more likely to continue using the tool for production code generation. But if the user finds problems with a code generator, they are unlikely to continue using it for production code generation. A log should be kept of problems found with the code generator, a small number of entries will help justify the use of the tool.
There are a number of annoyances that will cause users to be dissatisfied with a code generator, these include:

- Failure to detect incorrect configuration (either crashing or ignoring)
- Deterministic code generation. There should be no random behaviour in a code generator.
- Patches to the code generator should generate identical code where possible, so that it is easy for a user to verify that a patch has not produced any unwanted affects.

Verifying the results of AGC
Verifying the results of the automatic code generator involves ensuring the code is a correct implementation of the model. The techniques used for verification of manual code are mainly peer

review and testing, both of which are important and do find mistakes that need to be corrected, but are these techniques applicable to automatically generated code?

Peer Reviewing

Peer reviewing of manual code requires the code to be documented, structured and designed for maintenance, so that the reviewer can understand the code and thereby find errors. When code has been automatically generated, the author has little control on the structure and design of the code, but they do have control of the structure and design of the model. Therefore there is more to be gained from reviewing the model, ensuring that the model functionality is correct and the design is well engineered for maintenance, since this is the source for the automatically generated code. There is little point in reviewing automatically generated code for documentation, structure and design because an automatic code generator should always produce the same result, although the design of the code can be influenced by the design of the model. So a review of the code is mainly to find coding errors produced by the automatic code generator. An automatic code generator will not make the random types of faults that programmers make. But faults in automatically generated code will be systematic. The density of faults in code produced by an automatic code generator should be low, if the code generator works properly. If the fault density is anything less than low, the tool will not have any customer acceptance. So, since there should not be many faults in the code, the task of reviewing code will be very monotonous and if a fault was present in the code, the reviewer is unlikely to detect it, unless it was a gross error. Therefore there is little to be gained from peer review of automatically generated code.

One aspect of automatically generated code that will benefit from peer review is the interface of the automatically generated code to any manually written code or microcontroller registers. This is because the manually generated code is outside the control and structure of the automatic code generator, so the compatibility of the code will benefit from the inspection of a peer review to ensure correctness and compatibility.

Suppliers of automatic code generator tools do claim the code produced is readable, including comments allowing traceability to the block in the model that produced the code. This is still required to allow debugging and understanding of the code if an error is found in the code while testing, and to allow developers of the models to understand the code in the interest of improving the model configuration.

When writing code manually, one of the quality measures that is required of the code is that it conform to a sub-set of the high level language. There are a number of tools available that will check a high level language for conformance to a coding sub-set, but there is a question as to the validity of automatically generated code needing to conform to a sub-set. The answer is that most of the rules probably still apply. Some coding rules could be influenced by the way the model is configured to inform the code generator how the code should be produced, e.g. function breaks and variable scope, in which case the developer of the model has the ability to influence the conformance to the subset. Generally conformance to a subset should result in higher quality code and more user acceptance of the code generator.

Any complexity metrics that are collected for manually generated code are also applicable to automatically generated code. The complexity metrics are probably a function of the complexity of the model used to generate the code and therefore could be used as an indication to the maintainability of the model.

Testing

The testing of manually generated code has always been a fundamental method in ensuring that the code is correct, although frequently the testing performed on manually written code is to validate the functionality rather than to verify the code against it's design. In the case of automatic code generation against an executable model (e.g. Simulink) then it should be possible to validate that the model implements the requirements, and then verify the generated code against the executable model by dynamic testing. The testing of code against an executable model requires the execution of the model and code, compiled into an executable, with the same input stimuli followed by a comparison of the outputs and, possibly, any significant intermediate data. The comparison of intermediate data within Simulink would require the model to be instrumented, to extract the data, and the code may need to be changed to allow the data to be visible.

The comparison of the outputs has to be within acceptable error limits, unless the embedded code has been generated with 'double' floating point precision, which is unlikely, because most embedded controllers probably cannot afford the overhead of 64 bit floating point accuracy. There are 2 types of tolerance on the comparison of outputs, depending on the implementation of the code. If the code was produced using a fixed point implementation then the errors will be quantization errors which will be a multiple of the least significant bit of the output being compared, whereas if the code was produced using a floating point implementation, usually simply 'float' (32 bit floating point) then the error will be relative to the magnitude of the data.
The code can be compiled and executed on a PC, which is the simplest implementation for testing code, because the code can be compiled using a PC compiler and executed on the PC, this is known as Software In the Loop (SIL). The code can be compiled and executed on the target microcontroller, which requires the cross compiler and the embedded micro controller together with the infrastructure to update the input stimuli and extract the output data for comparison, this is known as Processor In the Loop (PIL). The advantage of PIL over SIL is that the embedded compiler and processor are also being tested, together with any effects on the size of 'int' for the embedded target.

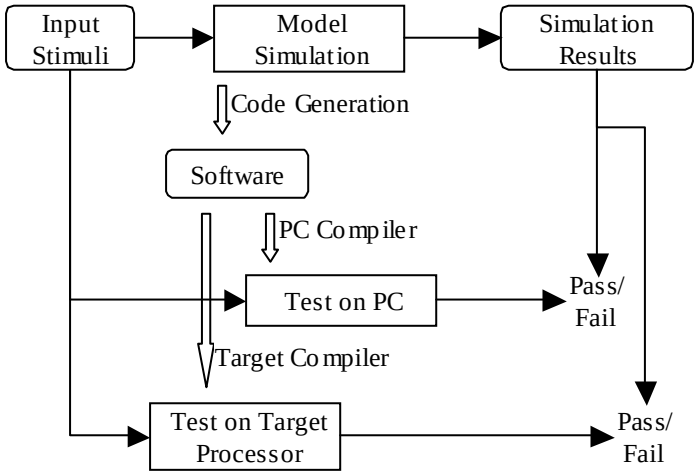

Figure 4: Process for testing automatically generated code

To allow the software to be tested the model needs to be modularised, so that testing can be done on a module-by-module basis. Although it is possible (and necessary) to test the whole system it is difficult to completely test the functionality of a sub-system when testing the whole system. Separating a system out into a number of modules also helps in understanding the system, and is a good software engineering practice to perform on a control system.

The report generated by the code generator needs to be reviewed; this may include errors, which should be fixed, and warnings, which may need to be fixed. The report from the code generator should be useful and allow the user to configure the model to allow the code generator to produce high quality code.

There are tools available that can perform 'dynamic' analysis on code, searching for programming errors in the code, these tools can also be applied to automatically generated code, although the tool cannot prove that the code functions like the model, but it does ensure that there are no arithmetic errors in the code.

Selected strategy for verifying and validating AGC
Of the methods presented earlier the scope for Land Rover to perform any validation of the code generator is very limited, due limited resources and the size of the task concerned. The validation of the code generator has to be considered as a code generator supplier's task, and evidence of an applicable process being followed would be all that Land Rover would request when selecting a code generation tool.
The primary strategy employed in verifying the automatically generated code is to perform testing of modules, endeavouring to prove that what behaves like the Simulink model 'is' the model. To be able to

prove that the code behaves like the model it is necessary to test all the model against all the code, therefore a test coverage criteria of 100% MCDC (modified condition decision coverage) is required, furthermore MCDC is a recommended coverage for testing automotive software [2]. It is necessary to apply this to the code, so that variable saturation can be tested (which will cause errors in the compared outputs). The code is instrumented to allow coverage metrics to be obtained using a standard code coverage tool. To facilitate the configuration of the input stimuli and comparison of the outputs a custom tool was written based around Microsoft Excel, which allows users to specify the values for the inputs and log the outputs while executing the model, then apply the input stimuli to the code and compare the outputs of the code to the previously logged values, any differences are high-lighted to the user. A custom tool was created that:

- Allows the user to specify un-related values to the input stimuli in order to achieve the required code coverage criteria more easily
- Save the test conditions consistently, allowing maintenance of the test cases by other users
- Facilitate regression testing on the model, if a model is modified then the old set of test cases can be run on the model to ensure there are no undesired modifications to the model

Code that interfaces to manually generated, base software is reviewed to ensure compatibility of the interface to the base software. Furthermore the automatically generated code is reviewed for conformance to MISRA-C, using a static analysis tool, dynamic analysis is performed to ensure that there are no 'bugs' and complexity metrics are also monitored for each module.

Technical problems implementing the strategy

The outputs of the Simulink model and generated code are usually not identical, because both floating point numbers and fixed point numbers are an approximation. At first impression a double precision floating point number is more accurate than a fixed point representation of a number, but this is not always the case. Consider a variable monitoring time accurate to 1 milli-second, as a double precision number this would be time, in seconds, and 0.001 as a double precision number is about 4.13E-19 too small, therefore when adding milli-seconds the error will also be accumulated and after 10 iterations it is less than 0.01. But a fixed point number representation could be scaled in milli-seconds and would be exactly 1, and after 10 iterations it would be 10, representing 0.01 exactly. If the model compared the variable to >= 0.01 and the variable was incremented by 0.001 every milli-second the double precision representation would be >= 0.01 after 11 iterations, whereas the fixed point representation would be >= 0.01 after 10 iterations [3]. Usually the floating point representation is more accurate than the fixed point representation, e.g. length in metres will be accurate to about 10^{-16} using double precision floating point, but will only be accurate to the chosen scaling, possibly milli-metres, or micro-metres using fixed point. In order to compare the results of the Simulink model and generated code it is necessary to introduce an amount of acceptable tolerance for each output. If the generated code uses fixed point scaling then the tolerance should be a multiple of the least significant bit's scaling. Whereas if the generated code uses single precision floating point then the acceptable tolerance should be a multiple of the data value of the output. Both of these tolerance schemes need to be implemented since the testing is being applied to different systems, one of which is fixed point and the other is floating point.

In order to achieve the code coverage it is necessary to create test cases for variables that 'saturate' to a maximum or minimum value, when the saturation occurs this will cause a difference between the 'perfect' model and the implementation. These differences will be out of tolerance and are highlighted as errors, but these are acceptable 'errors'. A special case of saturation is division by 0, where it is possible to require the code generator to check the divisor for 0 before attempting the division operation and return a different result. This is a reasonable check for the code, but in order to achieve the code coverage the test case needs to specify a divisor of 0, which, not surprisingly, causes Simulink to issue an error. A work-around for the division by 0 is to use values that are extremely close to 0, the quantisation errors of the fixed-point implementation allow the test for equality to 0 to succeed and Simulink will execute, although the differences between the model outputs and implementation outputs will again exceed the tolerance and are categorised as acceptable 'errors'.

Coding errors discovered using the validation strategy
The most common types of coding errors discovered using the validation strategy are type definition and scaling errors. These errors result in the code not being able to contain the data concerned, resulting in either an overflow, underflow or loss of precision. They are fixed by simply modifying the configuration of the variable storage for the data item concerned. In the early stages of code generation (i.e. the first time code is generated for a model) the number of scaling and type definition errors can be quite large, but these errors are quickly removed and the generated code is then tested more thoroughly.
Other errors that have been picked up by the testing strategy include:

- Incorrect re-use of data variables, where the model has been configured to use the same variable to store data from different blocks. This type of error is often quite difficult to find during system testing, but the technique of comparing the execution of the model to the execution of the code high-lights the problem when it is executed.
- Confusion between bit-wise and logical operations in StateFlow. The code generator interpreted C-like bit-wise operations as a bit operation, whereas by default StateFlow is configured to interpret C-like bit operations as logical operations.
- The testing strategy has found a few errors due to the automatic code generator creating incorrect code, it was simple to work around the problems by structuring the model slightly differently and the tool vendor was informed to enable the errors to be fixed in the next release.
- A few modelling errors have also been discovered with the Simulink models, due to visual inspection of the test cases.

The criteria for 100% MCDC coverage occasionally cannot be achieved due to unreachable code. The unreachable code can be created by the code generator, which represents inefficiency in the code generator's implementation of the model. This occurs every time a particular structure in the model is encountered and has to be accepted as a dispensation on the coverage criteria. The unreachable code can also be created due to constructs in the model, in which case it would also not be possible to execute that part of the model within Simulink. This represents an error in the model and the author would be able to correct the mistake.

Lessons learnt
It is often necessary to understand the structure and functionality of the generated code in order to determine where in the model it relates and how to increase the test coverage. Therefore the engineers involved in creating test cases need to have software engineering experience as well as modelling skills. The models that are being tested need to be designed for test. That is the control system needs to be partitioned into a structured architecture creating functional units that are small enough to be tested thoroughly. The exercise of partitioning a system into functional blocks is again a software engineering skill. Each of the functional blocks are partitioned into Simulink library models to allow maintenance and testing of the model by a team of engineers.

Process improvements
The testing that we have implemented is software in the loop. An improvement in this strategy would be to include the compiler and microprocessor, and run the test cases on code compiled for the target microprocessor. The infrastructure that we have will allow for this extension to the testing strategy, since the input stimuli and expected outputs are all specified using Excel, therefore it would be possible to extend this to compile the code for the target and run it on the processor.

Recommendations for the future
Thorough testing of the automatically generated code will need to be continued for the foreseeable future because of the configuration of the implementation, the variable types and scalings in particular. Even if the confidence in automatic code generation results in no code generation errors, there is always scope for a error to cause the system to mal-function due to the implementation configuration. So thorough testing will always be required, to prove that what behaves like a Simulink model 'is' the model.

Complete review of the source code is not justified because:

- The aspect of the review which checked against the requirements can be achieved by review of the model.

- The aspect of the review which checked for coding errors is not required as the random human errors will not be present.
- Static checking tools, e.g. adherence to language sub-sets and data analysis, can find model configuration errors.
- Systematic errors made when writing the code generator will be detected by thorough testing of the generated code.

However, a review of the interface between code generated by a tool and code produced manually is required.

It would be useful to determine the level of 'data' coverage that each data flow has achieved while running the tests. We currently monitor the code coverage, but in order to prove an algorithm that may not have many paths is implemented correctly it will need to be executed with more than one set of data.
We do currently do dynamic data analysis on the generated code, but this only checks for 'obvious' errors and does not compare the outputs to the expected outputs.

References

1. Michael Beine, Rainer Otterbach and Michael Jungmann, "Development of Safety-Critical Software Using Automatic Code Generation", SAE Technical Paper Series, 2004-01-0708, ISSN 0-7680-1319-4

2. Dr. Selim Aissi, "Software Testing Coverage for Automotive Embedded Controllers", SAE Technical Paper Series, 1999-01-1173, ISSN 0148-7191

3. Les Hatton, "Safer C: Developing Software for High-integrity and Safety-critical Systems", McGraw-Hill, 1994.