Building Trust in a Model-Based Automatic Code Generator

S. Tucker Taft
AdaCore, Inc.
81 Hartwell Ave #120
Lexington, MA 02421 USA
+1 781 750 8068
taft@adacore.com

Elie Richa
AdaCore, SAS
46 rue d'Amsterdam
75009 Paris FRANCE
+33 1 49 70 67 16
richa@adacore.com

Andres Toom
IB Krates
Mäealuse 4
12618 Tallinn Estonia
+372 680 5600
andres@krates.ee

ABSTRACT
If we wish to use an *automatic code generator* for the model-based development of a safety-critical system, how can we gain sufficient confidence in the correctness of the tool? For a tool like a code generator, which could *insert* an error into an airborne system, the US Federal Aviation Administration (FAA) requires the highest level of tool qualification [2], Tool Qualification Level 1 (TQL-1), if the tool is going to be used for a Level-A subsystem (one whose failure could be *catastrophic*). Achieving TQL-1 for such a code generator is analogous to achieving Level A certification for an embedded software component, but the lines of code in the tool can be substantially greater. In this paper we describe approaches to manage the complexity of specification and testing required for Level-1 qualification of a tool like an automatic code generator, a tool which includes multiple phases that transform an input model into optimized generated code.

CCS Concepts
• **Software and its engineering~Automatic programming**
Software and its engineering~Software testing and debugging
Software and its engineering~Formal software verification.

Keywords
Automatic code generator, Simulink, Tool qualification, TQL-1, DO-178C, DO-330, Integrated unit testing

1. Introduction
If we wish to use an *automatic code generator* for the development of a safety-critical system, how can we gain sufficient confidence in the correctness of the tool? For example, given a code generator that takes a real-time model for a flight-control system represented in Simulink® and Stateflow®, and turns it into MISRA C or the SPARK subset of Ada, what process could ensure that the generated code is a faithful representation of the original real-time model? The US Federal Aviation Administration (FAA) has a well-defined process for creating a *qualified* code generator, meaning a code generator whose output can be trusted to match exactly the semantics of the input model, with nothing left out, and nothing added. This process is defined in DO-178C (Software Considerations in Airborne Systems), and its accompanying documents DO-330 (Software Tool Qualification Considerations) and DO-331 (Model-Based Development and Verification) [5].

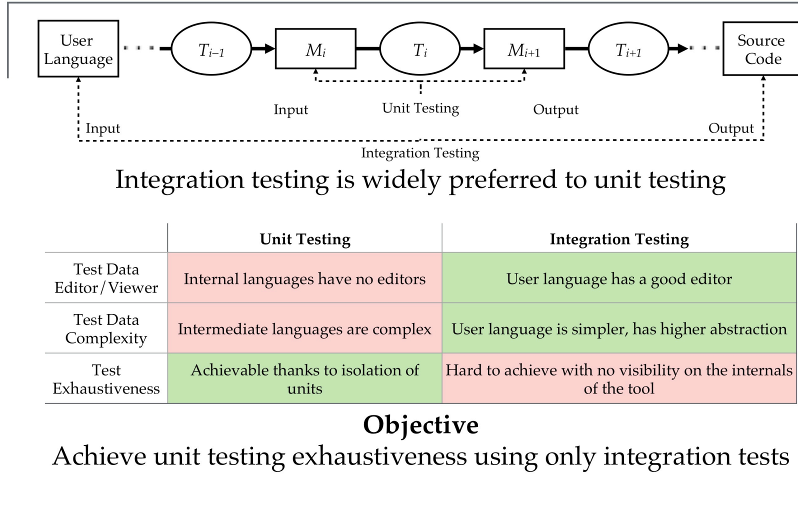
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HILT 2016, October 2016, Pittsburgh, PA, USA.
Copyright held by the authors.

For a tool like a code generator, which could *insert* an error into an airborne system, the highest level of tool qualification [2], Tool Qualification Level 1 (TQL-1), is required if the tool is to be used for a Level-A subsystem (one whose failure could be *catastrophic*). Not surprisingly, this level of tool qualification can involve a great deal of time and effort, often estimated in the hundreds of hours per KSLOC (thousand source lines of code) of the tool. This is similar to the level of effort, per line, required for verifying a level-A safety-critical embedded software component, but tools can be significantly more lines of code. For example, if the tool were 100KSLOC, the traditional approach to verification at level A might cost in the ballpark of five million USD. Hence, there is a strong incentive to investigating alternative approaches to testing such a tool, while still achieving the TQL-1 objectives.

2. Traditional Approaches to Testing
The traditional approach to verifying a high-integrity application involves carefully defining and validating a set of high level requirements for the application, then deriving lower-level, module-level requirements from the high-level requirements, which are specific enough to determine the appropriate implementation, and then to check each module of the implementation against its low-level requirements using unit testing, and finally to perform integration-level testing of all high-level requirements. Coverage analysis would be performed to ensure that all code is covered by these tests, and to ensure there is no code remaining in the application that might provide extra, undesired functionality.

For an embedded software component, this combination of unit-level testing of each module and integration-level testing of the component as a whole can work well. In particular, unit-testing of embedded-software modules is practical because in many cases the number and complexity of inputs for each module are manageable, and the outputs are relatively easily identified and checked. However, for a tool like an automatic code generator, which generally involves multiple phases involving progressive transformation of the input model into the generated code, unit testing can be a real challenge. On the other hand, integration testing is not significantly harder for such a tool, as the number of intermediate phases does not affect the overall inputs and outputs of the tool.

This dichotomy between the complexity of unit testing and the relative ease of integration testing of a multi-phase tool like a code generator is illustrated in Figure 1.



	Unit Testing	Integration Testing
Test Data Editor/Viewer	Internal languages have no editors	User language has a good editor
Test Data Complexity	Intermediate languages are complex	User language is simpler, has higher abstraction
Test Exhaustiveness	Achievable thanks to isolation of units	Hard to achieve with no visibility on the internals of the tool

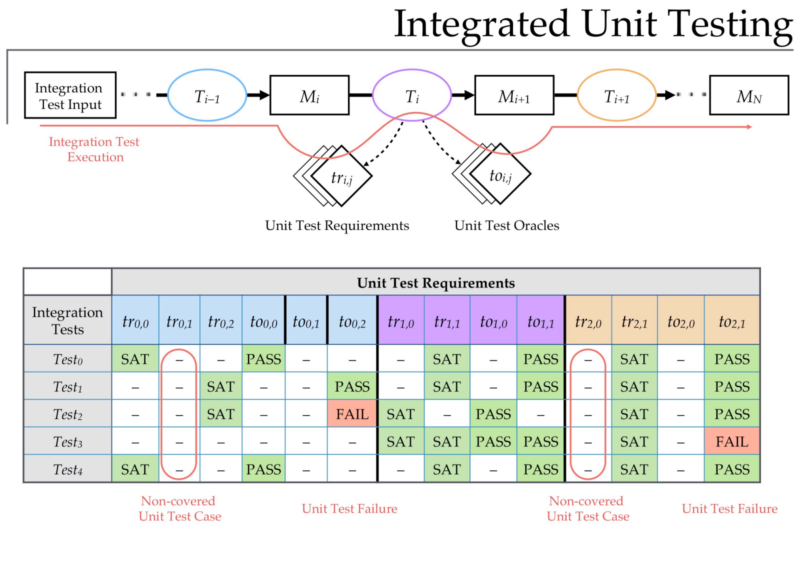
Figure 1

In Figure 1, we show the overall data flow of an optimizing, automatic code generator, where the input model is referred to as the "User Language" and the output (generated code) is referred to as the "Source Code." Multiple phases are pipe-lined, with the first phase reading in the original model represented in the User Language (M_0), and representing the model in some internal data structure M_1. This is then transformed progressively into lower level representations of the model, M_2, M_3, etc. which get closer to code level, until finally the final phase produces actual Source Code in the desired programming language. To perform integration testing, one only need prepare a model represented in the User Language, using the normal model creation tools, feed it through the code generator, and then examine the generated Source Code to determine whether it satisfies the high-level requirements in terms of form and functionality, using normal compilers, static analysis, and testing tools for that programming language.

By contrast, performing unit testing of each of the phases of such a multi-phase code generator is significantly more complex, in that an internal data structure must be constructed for each test of a given phase that conforms to the representation used for input to that phase, then the phase needs to be invoked on that input, and then the output representation must be checked to see whether it has the expected form and content. Preparing such inputs, and checking such outputs, requires laborious manual processes or the creation of special tools, which themselves might need qualification at least to some level.

3. Integrated Unit Testing

Given the complexity of unit testing, an alternative approach has been developed which has been dubbed *Integrated Unit Testing* [4][6]. This approach is illustrated in Figure 2.



	Unit Test Requirements													
Integration Tests	$tr_{0,0}$	$tr_{0,1}$	$tr_{0,2}$	$to_{0,0}$	$to_{0,1}$	$to_{0,2}$	$tr_{1,0}$	$tr_{1,1}$	$to_{1,0}$	$to_{1,1}$	$tr_{2,0}$	$tr_{2,1}$	$to_{2,0}$	$to_{2,1}$
$Test_0$	SAT	–	–	PASS	–	–	–	SAT	–	PASS	–	SAT	–	PASS
$Test_1$	–	–	SAT	–	–	PASS	–	SAT	–	PASS	–	SAT	–	PASS
$Test_2$	–	–	SAT	–	–	FAIL	SAT	–	PASS	–	–	SAT	–	PASS
$Test_3$	–	–	–	–	–	–	SAT	SAT	PASS	PASS	–	SAT	–	FAIL
$Test_4$	SAT	–	–	PASS	–	–	–	SAT	–	PASS	–	SAT	–	PASS

Figure 2

In Figure 2, we show a process that embeds unit-test-requirement monitors and unit test *oracles* (a checker which "knows" what is the desired output), directly into the structure of the tool. With these monitors and checkers embedded in the tool, we then do the steps used for normal integration testing, preparing representative models ($Test_0$ through $Test_4$) and feeding them through the code generator. But now, rather than merely waiting for the tool to generate the final output, each embedded unit-test requirement monitor keeps track of whether an input to its associated phases matches its associated unit-test, and if it does, it logs that fact and then triggers the action of a corresponding unit-test oracle-based checker upon completion of the phase, which verifies that the output of the phase corresponds to the expected transformations of the input, for the particular test pattern.

For example, imagine we have defined a particular transformation of a "gain" block at the model level into an expression at the code level that multiplies the value of a signal variable by a constant. We would have a unit-test-requirement monitor logging every time a gain block shows up in its model-level input representation, and when it does, trigger the oracle-based checker to look at the code-level output representation to be sure it involves a multiplication of the appropriate signal variable

by the appropriate constant. This is a very simple check to perform, and so long as enough models are passed through the tool as a whole, coverage of this particular unit-test pattern can be expected.

After running a number of models through the tool, we can end up with a table like the one in the above diagram. Along the left side we have the models, $Test_0$ to $Test_4$. Along the top we have the pairs of test requirement and test oracle, for each distinct phase of the tool. For example, $tr_{0,2}$ means the test requirement 2 for phase 0, while $to_{2,1}$ means the test oracle 1 for phase 2. Each time a particular input to a phase, coming from a given model, *satisfies* the test pattern associated with some test requirement, we will see a SAT in the requirement's column at the input model's row. Each time a test oracle is invoked we will see either a PASS or FAIL in the oracle's column at the input model's row. If we ever end up with an empty column, that implies that the test pattern was never encountered (the corresponding low-level requirement was not "covered"). If we end up with a FAIL in a test-oracle column, that means we have a test failure (the corresponding low-level requirement was not properly implemented). In the above table, we see that $tr_{0,1}$ and $tr_{2,0}$ were not covered, while $to_{0,2}$ and $to_{2,1}$ had failures. Such a table documents a thorough unit testing process while avoiding the expense of preparing special inputs for each test pattern.

Note that *coverage* of the input test patterns implies coverage of the set of requirements, it does not imply *coverage* of the tool's source code. Separate source-code coverage tools are used for assessing whether all statements and conditions in the tool are exercised by the test suite. If not, this may indicate that a requirement is missing, implying the need for additional input patterns and oracles. Alternatively, it may indicate that certain parts of the code are not applicable in normal execution of the tool, and instead are used for debugging, or for non-standard modes of use. Any such unexercised code requires separate justification as part of qualifying the tool under DO-178C [5].

One concern comes from the case where one transformation can create the opportunities for additional transformations, perhaps involving repeated invocations of the same phase. This can be accommodated with this methodology, because the individual oracles check the correctness of the transformation of any single recognized input pattern. They do not preclude the output of the transformation being fed back through the same phase again, provided it again passes through the input pattern check.

4. Factored Specification of Requirements

To identify the interesting test cases, and to create the test oracles required by this Integrated Unit Testing approach, traditional manual specification of requirements and test cases for each interesting transformation are labor intensive and also error prone. The example of the "gain" block given above is just one of over one hundred blocks of interest for an automatic code generator, and typically each block has many variants to suit various engineering needs. As an example, the "gain" block can turn into a scalar times scalar multiplication, a scalar times vector, or a scalar times matrix multiplication. Furthermore, the types involved could be integers or floating point values of various sizes and the calculations could be subject to different rounding and saturation options.

All of these gain-block cases are potentially interesting, but the actual implementation of the transformations is factored so that the gain-block-specific transformation worries only about the insertion of a multiplication operator into the lower-level representation, and relies on later phases to expand such an operator as appropriate to deal with scalar, vector, or matrix computations of various types. Given that the implementation is itself factored, it makes sense that the specification of requirements should similarly be factored. Unsupported configurations must be properly rejected and result in corresponding robustness requirements and test cases.

Factoring the requirements is based on representing the possible variants of each block as a tree (or directed graph) structure, using a formal *Block Library Specification* (BLS) language, where alternatives such as scalar vs. vector and integer vs. floating point are represented as branching points in the tree. Requirements associated uniquely with a particular branch of the tree of variants are connected at that point, meaning that the set of requirements for a specific variant, such as a scalar-times-vector floating-point gain block, are derived by walking the tree down to a particular leaf, gathering the requirements along the branches visited [3].

Given the factored requirements, it is possible to identify all interesting test cases, and corresponding oracles, by walking these tree representations. With appropriate tooling, the Integrated Unit Testing code to insert on either side of each transformation phase can be generated from these trees represented in the BLS language, rather than by tedious and error-prone manual coding, further reducing the expense of achieving adequate testing.

5. Related Work

There is one commercial model-based code generator that has already achieved the TQL-1 level of qualification -- the ANSYS SCADE code generator [7]. SCADE is based on LUSTRE, a more formally defined language than Simulink, yet the DO-178C qualification of SCADE is also based on a testing approach [8]. Because of the way DO-178C is defined, a largely testing-based approach to qualification is currently the only practical alternative [5]. Nevertheless, work has been done to create a LUSTRE-based code generator that can be proved formally to produce generated code that is equivalent to the input model [9].

Developing a formally certified code generator for Simulink would depend on having a more formal definition of the Simulink semantics. The ClawZ system [10], which can translate a subset of Simulink to the Z language, could be the basis for such an effort. The approach taken in [10] and similar work in [11] falls into the general alternative for verifying the correctness of a translator by verifying the equivalence between the generated code and the input model using a separate tool, which is run each time the translator is run. To satisfy DO-178C requirements, this technique relies on the independence of the separate tool, and the qualification of the separate tool to a somewhat lower level of safety, namely TQL-4. TQL-4 may be used, because this tool is not itself generating the code, and it cannot itself insert errors, though it can fail to identify errors inserted by the translator. However, formally proving that the generated code correctly refines the specification is non-trivial. It is reported in [10] that in the ClawZ system many of these proofs can be automated. However, in general, this is a challenging task and depends on the complexity of the generated code. Secondly, this approach is focused on proving the correctness of the generated code with respect to its specification, rather than the correctness of the code generator itself. The correctness of the specification must still be independently verified.

The CompCert compiler is an example of a formally certified multi-phase translator, in this case from C to assembly language [12]. In the long run a formally certified translator for Simulink

seems feasible. However, the approach is far from trivial and the co-development of formal specifications, proofs, and program code raises significant “common mode“ concerns from the DO-178C viewpoint, which must still be sufficiently addressed. Also, the DO-178C process is currently still heavily based on testing, and because each project making use of a “qualifiable” tool must actually “qualify” the tool in the particular environment in which the tool is used, a thorough testing-based strategy remains essential to meeting the DO-178C requirements for qualification [5].

6. Conclusion

Building trust in a code generator is essential if we are going to rely more and more on such tools to help automate the generation of safety-critical software from higher-level models. However, innovative approaches are needed to manage the potentially prohibitive expense of achieving tool qualification for a modern, optimizing code generator, at the highest level of trust, TQL-1. Integrated Unit Testing is one such approach. When combined with the systematic, factored approaches for specifying requirements formally, and generating components such as requirement monitors and oracles from these requirements, it becomes possible to achieve TQL-1 in a way that not only is more cost effective, but also supports incremental qualification as the tool evolves. AdaCore is in the process of qualifying its QGen automatic code generator using these approaches [1].

7. Acknowledgements

We thank the anonymous reviewers for their helpful comments. This work represents the joint work of AdaCore and IB Krates, and was funded in part by the French and Estonian ministries of research, industry, and defense through the Projet-P, Hi-MoCo, and Vorace projects. The factored specification of requirements is based largely on work documented in [3], led by Arnaud Dieumegard.

8. References

[1] AdaCore, QGen Model-Based Tool Suite, http://adacore.com/qgen.

[2] Certification Authorities Software Team (CAST), CAST-25, “Considerations when using a Qualifiable Development Environment (QDE) in Certification Projects,” FAA, Sep 2005, https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-25.pdf.

[3] Dieumegard, A. *et al*, Block Library Driven Translation Validation for Dataflow Models in Safety Critical Systems, FMICS-AVoCS 2016, Pisa, Italy, Sep 2016.

[4] Richa, E. *et al,* Towards Testing Model Transformation Chains using Precondition Construction in Algebraic Graph Transformation, Third Workshop on the Analysis of Model Transformations, AMT’14, Valencia, Spain, Sep 2014, http://ceur-ws.org/Vol-1277/4.pdf.

[5] Rierson, L., Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance, CRC Press, 2013.

[6] S. T. Taft, “TQL-1 Qualification of a Model-Based Code Generator,” HCSS 2016, Annapolis, MD, May 2016, http://cps-vo.org/node/24503.

[7] ANSYS Esterel Technologies, “SCADE Suite® KCG 6.4 DO-178C Certification Kits Technical Data Sheet,” http://www.esterel-technologies.com/wp-content/uploads/2013/02/SCADE-Suite-DO178C-Certification-Kit.pdf

[8] ANSYS Esterel Technologies, “Efficient Development of Safe Avionics Software with DO-178C Objectives Using SCADE Suite®,” http://www.peraglobal.com/upload/contents/2015/11/20151113142739_85462.pdf

[9] Biernacki, D. *et al*, “Clock-directed Modular Code Generation for Synchronous Data-flow Languages,” *LCTES’08*, Tucson, AZ, June 2008, https://www.di.ens.fr/~pouzet/bib/lctes08a.pdf

[10] O’Halloran, C., “Automated verification of code automatically generated from Simulink®”, *Automated Software Engineering* 20(2):237-264, June 2012

[11] Ryabtsev, M. *Translation validation: from Simulink to C*. Diss. Technion-Israel Institute of Technology, 2009, http://ie.technion.ac.il/~ofers/publications/theses/Michael-Ryabtsev.pdf

[12] Leroy, X., Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107-115, 2009.