An Assessment Framework for Automatic Code Generator Tools

Neil Audsley; Department of Computer Science; University of York; UK

Iain Bate; Department of Computer Science; University of York; UK

Colin O'Halloran; QinetiQ; Malvern Technology Centre; Malvern; WR14 3PS; UK

Keywords: autocode generation, metrics, software

Abstract

This paper addresses the assessment Autocode Generators (AG) for their suitability for use in the development of software for critical systems. The assessment methods presented are independent of how the AG was designed; therefore it is applicable to both white box (bespoke) and black box (COTS) AGs. The paper defines a number of attributes that relate to the quality of generated code from an AG and the cost effectiveness of using the AG. A number of metrics are derived to evaluate an AG against the attributes. For evaluation purposes, these metrics are applied to a number of COTS AGs.

Introduction

Autocode Generators (AGs) are often considered for development due to their perceived cost advantages in the production of software. An AG enables systems to be developed at a higher, model-centric level, with automatic translation of the model into software written in some high-level language. Other perceived benefits of AGs include: increased (and more consistent) quality, design related issues addressed at a higher level, and removal of discontinuities between disciplines.

For use in the safety related and safety critical domain trust in the generated code must be established. Existing arguments have relied on knowledge of the internals of the AG (i.e. white box solutions) [1, 17]. For COTS AGs such knowledge is not usually available.

This paper establishes a framework for assessing AGs, concentrating upon the actual software produced. Note that the assessment framework applies to any AG, including COTS. The paper demonstrates the application of the metrics by evaluating for the commercial AG tool Simulink. Evaluations of other tools (Ilogix's Statemate and UML Scriptor) are only summarised for space reasons.

Previous Work:

In [1], Whalen and Heimdahl established five requirements for high integrity code generation:

1. Source and Target languages must have a formally well-defined syntax and semantics.
2. The translation between a specification expressed in a source language and a program expressed in a target language must be formal and proven to uphold meaning of the specification.
3. Rigorous arguments must be provided to validate the translator and/or the generated code.
4. The implementation of the translator must be rigorously tested and treated as high assurance software.
5. Generated code must be well-structured, documented and traceable to the specification.

It is clear that COTS AGs struggle to meet these requirements, particularly 2, 3 and 4.

In [2], three basic approaches are identified for assessing AG quality:

1. Verified autocode generator
2. Correctness by construction (Autocode synthesis)
3. Verification of produced code

Verified AGs are extremely difficult to produce. The analogy to verified compilers is drawn. To ensure correct compilation, formally verified compilers have been proposed [3]. Such compilers are proved, in advance of any compilation of source code, to always produce object code that correctly implements the source. Verified compilation has proved extremely difficult for realistically sized high-level languages. Indeed, Bowen states that *"in the short term, it seems unlikely that a production compiler will ever be completely verified"* [4].

Automatic code synthesis based on automatic proof is dependent on the soundness of the theorem prover. It is directly responsible for the generation of the code. This implies a verified theorem prover is required, this is still commercially difficult to justify.

This paper addresses the third approach, that of automatic generation and then separately assessing the resulting code.

Paper Overview:

Clearly, the use of AGs (particularly COTS AGs) for the production of software for use in safety-critical systems requires the verification of the produced software[1]. This paper considers part of this verification, namely the assessment of generated software in order to establish its quality. This can be subdivided into: ability to assess that the resulting software meets the specification; and establishing whether the AG itself is fit for purpose.

The paper proceeds by defining appropriate attributes [6], defining the relationships between the attributes, and showing how the attributes relate to systems that are produced using AGs.

Attributes for Gathering Verification Evidence for the Certification Case

The aim of the software safety case is to show that the risk of hazards associated with systematic failures is acceptably low. There are a plethora of standards that exist related to the certification of software with each standard being applicable to different domains, applications, countries etc. However as part of producing the report "*Guidance for the Use of Ada in High Integrity Systems*" [8], a rapporteur group (Annex H Rapporteur Group - HRG) categorised the forms of verification to be performed that are common across most standards as part of defining appropriate language subsets, not just Ada. Therefore to avoid consideration of the individual features of standards and their demands for assessment of software in critical systems, the attributes for assessment are taken from the Ada HRG report.

The forms of verification are summarised in Table 1 and Table 2 in appendix 1, along with a discussion of their relationship to AGs. The first two columns in the tables identify the type of analysis and provide a description of what the aim of the resulting evidence is. The third column discusses issues related to the use of an AG if the analysis is to be performed. The fourth column indicates whether the verification is actually needed if other evidence is available that the AG can always be trusted to meet the requirement. The justification is given in italics – e.g. data use analysis is not needed if the AG is shown to uphold write before read. However in practice, evidence needed on a per-use basis may be weakened each time the AG is used, as confidence grows in the output that it produces. In general, the verification is only still needed if the evidence cannot be collected at the source code level, e.g. timing analysis. AGs that meet the criteria have the distinct advantage that the evidence need only be gathered once rather than on a per-use basis. In addition, Table 3 contains extra attributes for assessment which are demanded by most standards - review and traceability.

Key points that emerge from Tables 1-3 with respect to a number of the forms of verification are discussed in the following subsections.

[1] Clearly, the effort required to perform the verification should not negate the benefits of having an AG. This suggests that the verification should be mostly automated.

Verification Technique	Description	Issues when Gathering Evidence for AG Solution	Needed with trusted AG?
Control flow analysis	Ensure code is executed in correct order and that it is structurally well formed.	Ensuring correct order via automated means often relies on software being annotated with required order [7].	× (*AG ensures requirements are met.)*
Data flow analysis	Ensure no variable can be accessed before it has been set.	Set before use may not have been implemented in AG.	× (*AG's construction would guarantee set before get.)*
Information flow analysis	Identifies relationships between inputs and outputs in-order to ensure the correct dependencies are implemented and no other dependencies exist.	Checking dependencies via automated means often relies on software being annotated with required dependencies [7].	× (*AG guarantees dependencies were met and no new ones introduced.)*
Symbolic execution	Verify properties of the software, without resorting to formal proofs and proof tools.	Coding standard followed by AG may hinder ability to symbolically execute resulting code.	× (*AG ensures requirements are met.)*
Formal code verification	Proving that the code is correct with respect to a formal specification.	Coding standard followed by AG may hinder ability to formally prove resulting code. Often relies on software being annotated. [7].	× (*AG ensures requirements are met.)*
Range checking	Verify that data values are within a specified range.	AG may not produce code in a strongly typed language which makes range checking difficult.	× (*AG ensures requirements are met.)*
Main memory usage	Determine the maximum execution time for tasks and that execution times are bounded.	Dependent on the compiler and processor as well as support from the coding standard followed by AG.	✓ (*Memory usage not totally reliant on AG but influenced by it.)*
Stack usage analysis	Determine the maximum stack usage used and show that the processor has sufficient stack memory.	Dependent on the compiler and processor, not the AG.	✓ (*Stack usage not dependent on AG.)*
Timing analysis	Determine the maximum execution time for tasks and that execution times are bounded.	Dependent on compiler and processor as well as support from the AG's coding standard.	✓ (*Same as for Main memory.)*
Other memory usage	Determine worst-case other memory usage (e.g. communications buffers) and whether the system has sufficient capacity.	Dependent on compiler and processor as well as support from the AG's coding standard.	✓ (*Same as for Main memory.)*
Object code analysis	Determine the object code upholds the source code intent.	Dependent on the compiler and processor, not the AG.	✓ (*Not dependent on the AG.)*

Table 1 – Summary of the Ada HRG Analysis Requirements in the Context of an AGs

Verification Technique	Description	Issues when Gathering Evidence for AG Solution	Needed with trusted AG?
Equivalence class	Given exhaustive testing is impractical, equivalence class testing divides the input and output data spaces into classes so that a test with any of the values in one class should give equivalent results.	Requires test scripts to be produced which means understanding of how the code is produced is important.	× (*AG ensures requirements met.)*
Boundary value	Builds on the equivalence class of testing to test at the boundaries of value classes, rather than just at some point in the class.	Same as for equivalence class	× (*Same as for equivalence class)*
Statement coverage	Apply test cases so that each program statement has been invoked at least once.	Same as for equivalence class	× (*Same as for equivalence class)*
Branch coverage	Apply test cases so that each decision in the program has taken each of its possible outcomes.	Same as for equivalence class	× (*Same as for equivalence class)*
Modified condition / decision coverage	Show that each basic condition independently affect each basic decision.	Same as for equivalence class	× (*Same as for equivalence class)*

Table 2 – Summary of the Ada HRG Testing Requirements in the Context of an AGs

Verification Technique	Description	Issues when Gathering Evidence for AG Solution	Needed with trusted AG?
Traceability	Showing how and where requirement(s) are satisfied in the code.	Potentially more difficult since transformation process is unknown and often not intended to be reversible.	× (*Less important since requirements guaranteed as met.)*
Reviews	Checking the output of individual process stages, e.g. verification.	Potentially more difficult since transformation process is unknown.	× (*Same as for Traceability.)*

Table 3 – Summary of Other Requirements in the Context of an AGs

Coding Standard:

The suitability of the coding can be broken down into three broad areas; general suitability, specific suitability and ability to configure the coding standard. These are discussed in the following subsections.

General Suitability:

The most important issues related to the coding standard are:

- whether the coding standard can be determined;
- whether the resulting software obeys commonly accepted coding requirements (e.g. no unstructured control flow, bounded loops and recursion, writing the value of variables before reading their value).

Specific Suitability:

This section discusses the ability of an AG to meet specific coding standard requirements automatically. Such requirements could be diverse; from optional items (e.g. the "best" way to configure loops) to conforming with a particular language subset (e.g. SPARK Ada [7]) or providing annotations to support information flow analysis.

It is unlikely that such specific requirements are met in a COTS AG, unless the AG is specifically procured to meet such requirements. Cost-benefit issues for such an "enhanced" AG include:

- A highly complex AG will undoubtedly be expensive to procure and qualify.
- A highly specific AG would be expensive because it would be a one-off development.
- Dependence on the characteristics of the generated software that might have a significant impact on the system and software architecture. For instance, if the AG produces conventional function-oriented software and there is a need to use this software with existing software in an object-oriented framework, then the generated code may have to be re-structured when generated or compromises made, e.g. interfaces to handle the paradigm shift, in the framework. Similar re-structuring or compromises may be needed if the system adopted an Open System Standard such as ARINC 653.
- Whether the automatically generated code supports a particular verification plan (e.g. the use of the SPARK Examiner to perform information flow analysis), if not the generated code would have to be altered or the verification plan compromised.
- A project may specify a particular language (or subset). If the AG does not produce code conforming to the requirements, then the generated code may have to be altered after it is produced such that it does conform; thus increasing cost.

Configurability:

Rather than trying to produce or procure a specific AG as discussed in the previous section, a more versatile solution is to obtain a configurable AG. This could include some of the following features:

- Ability to alter the templates used to generate the code so that the correct language subset is used and/or to make verification easier.
- Ability to augment the templates with annotations that can help with verification or traceability.
- Ability to alter the templates so that a specific interface (e.g. the API of ARINC 653) is generally adhered to.
- Ability to alter the way the code is generated so that a specific interface (e.g. the API of ARINC 653) is adhered to in a certain region of the generated code in order to make interfacing with legacy code more straight forward.

Automatic Test Generation:

An issue for any user of an AG where the need to perform unit testing has not been removed by the AG being considered trusted, there is the need to show the system meets its requirements. Clearly if this involves producing test scripts as part of coverage assessment or unit testing, then the ease with which this is performed can significantly affect the cost benefit of using the AG. The ideal solution is that the framework also supports the automatic independent verification of the generated code.

Impact of Compiler and Processor:

A number of forms of verification, e.g. timing analysis, are more dependent on the compiler and processor than the AG. However their impact cannot be ignored since the code that is input to them can affect the ability to perform the verification – e.g. the nature of the source code input to a compiler can affect the accuracy of timing analysis. In these cases, the nature of the source code needed or preferred should be contained in the minimum-coding standard dictated of the AG.

Attributes Related to the Environment

Earlier in the paper considered issues that are quantitative in nature. In contrast, this section considers qualitative environment issues, concentrating upon whether an AG is fit for purpose.

Attribute	Description	Issues for AG Solution	Application Dependent
Hand Tailoring	The degree of hand tailoring the generated code will need to fit in with the application's framework.	If too much hand tailoring is needed and/or the generated code is hard to understand, then the benefit of using an AG will be diminished.	✓ *(The generated code needs to fit in with application framework.)*
Other Code	Ease with which code not generated by the AG (e.g. human-produced code) can be integrated with code generated by the AG.	Same as for *Hand Tailoring*.	✓ *(Generated code needs to fit in with existing application code.)*
Sufficiency	The degree to which the AG can generate code for the specification language, i.e. percentage of language constructs and combinations.	Where the AG is not sufficient, then these cases need to be clearly identified. Issues for dealing with cases insufficient are the same as for *Hand Tailoring*.	×
Correctness	The degree to which the AG can generate code satisfactorily (i.e. ability to argue integrity of the output) for the specification language.	Where the AG is not satisfactory, then these cases need to be clearly identified. Issues for dealing with cases insufficient are the same as for *Hand Tailoring*.	×
Flexibility	The versatility of the tool to support a range of application types.	A less flexible tool is likely to be simpler and hence more likely to be demonstrable as correct.	×
Cost of Evidence	The relative cost (relative might be compared to a trusted AG, another tool or hand-crafted code) of gathering evidence that generated code is fit for purpose.	Where it can be influenced, there is a trade-off between building the AG so it can be trusted (wholly or in certain verification areas) and lowering the cost of developing of the AG.	✓ *(The cost is dependent on the integrity requirement.)*
Ability to Qualify	Whether the AG could be qualified for use on a project or range of projects. This is related to how trusted the AG might be.	Same as for *Cost of Evidence*.	✓ *(Benefit of qualifying the AG is dependent on integrity requirement.)*
Assessment versus Operational Experience	Whether the integrity evidence for the generated code, or the AG itself, is based on assessment or operational experience.	If an AG is not initially trusted, with use the confidence in its operation will be established leading to a reduced need to check its operation	✓
Model versus Formal	Whether the tool adopts a formal approach or a more traditional model based approach.	The ideal option is often viewed as the AG and model development environment having formal underpinnings transparent to the user.	×
Assess Tool versus Assess Code	Whether the integrity assessment is a one-off or per-use exercise.	Same as for *Cost of Evidence*.	✓ *(Ability to make one-off arguments dependent on the integrity requirement.)*
Completeness of Verification	Whether the verification evidence provided by the AG automatically is complete.	Same as for *Cost of Evidence*.	✓ *(Verification needed dependent on the integrity requirement.)*
Maintainability	The ease with which code output from the tool can be maintained.	If the generated code is hard to understand or small changes in the specification lead to un-proportionally large changes in the code, then the benefit of using an AG will be diminished.	×
Repeatability	Critical systems standards may expect tools producing part of the product, e.g. compiler, to be able to repeat an operation.	Preferably the AG should be configurable to ease this problem.	×
Safety Argument	Whether the use of an AG is detrimental to the ability to certify an application that uses the generated code.	Can it be justified the AG cannot lead to errors in the code and hence system-level hazards. Benefit could be attained by separately arguing about the requirements model from the code generation.	✓ *(Dependent on the integrity requirement.)*

Table 4 – Attributes for Environment

Table 4 presents the list of attributes relating to the environment in which the AG is deployed. The table provides a description of what the attribute means, the relationship of this attribute to the AG, and an indication of whether the attribute is dependent on the nature of the application. Some of the importance ratings will be dependent on the context of their use. For example, flexibility is not an issue if the AG is only required for one type of application. The dependency on application refers to situations where an AG might be trusted (i.e. no further evidence need be generated) for lower integrity applications but not for higher integrity applications. Where it is felt it might be needed, explanations of why an attribute is/is not dependent on the application are given in *italics*.

Key points that emerge from Table 4 with respect to a number of the forms of verification are discussed in the following subsections.

Modifying Generated Code for Intended Context:

As previously discussed, a significant issue is whether generated code needs to be altered after generation. If alteration is required, the cost benefit of using an AG may be diminished to a point where it becomes better to produce code in a conventional manner. Therefore, when selecting an AG for a particular project, it is important to consider whether generated code would need to be altered to be consistent with the particular verification / certification approach adopted for the project [2].

AG Qualification and Safety Argument:

Assessment versus Operational Experience:

Dependent on how the AG is obtained the approach to how we justify its integrity will alter. The following options can be used independently or in combination:

- argue its correctness based on the way it is constructed and subsequent testing of its operation;
- or argue confidence in its operation based on use in similar or appropriate contexts.

For AGs produced in-house or with a cooperating partner where the AG is a white box entity, then the first option is preferable. Otherwise, the only realistic approach is the second option that might mean constructing test rigs to allow extensive of the AG.

Simplicity versus Flexibility:

A cost trade-off that needs to be performed is whether the AG obtained is simple or flexible. A simple AG may not be usable for many different applications but should be easier to gather evidence for the generated code. An example of this type of AG is SCADE [9] that is mainly used for generating control systems such as the A340 Flight Control System. Whilst the approach behind this tool and the models, from which code can be generated, are relatively simple, it has been used in one of the highest integrity applications which is certified to Level A DO-178B. Tools such as Simulink [10] have much more versatile AGs, however to date to the best of our knowledge, these tools have not been used to auto generate code for high integrity applications being certified in the UK.

Evidence at the Higher Level:

A key issue with the use of AGs is where and how safety is argued and supporting evidence collected. This discussion is beyond the scope of this paper.

Verification

One-Off versus Per-Use:

An issue is whether the AG can be verified as a one–off activity, whether its output has to be verified every time it is used or more likely a combination of the two. Clearly, the first of these options is preferred. Alternatively, the third option should be adopted with the amount of per-use verification being reduced as confidence in the AG is increased. Similar to how the choice of compiler is argued, the approach followed depends on the type of certification strategy followed, and the time and money available to producing the AG. The latter of these would form part of any trade-off analysis performed when producing the system.

Completeness of Automatic Verification:

Similar to issues related to having to modify the generated code, there are issues regarding having to produce separate test scripts (or at least modify those automatically produced) in-order to produce the necessary evidence. These issues relate to degrading the benefit of using an AG. They should be considered when choosing or specifying an AG as part of any trade-off analysis.

Evaluation Method

This section presents an evaluation approach that covers the key attributes discussed in the previous sections. A key feature of the evaluation is the lack of reliance on any one form of evidence. Instead, it draws on information from a number of sources.

Software Complexity

The qualitative and quantitative assessment of software complexity can be used as part of evaluating a number of the attributes, including; traceability, maintainability, reviews, control flow analysis, all forms of testing, hand tailoring, other code, and cost of evidence. The qualitative assessment is performed by inspection of the specification and code. The quantitative assessment is performed by measuring the code using two well-known metrics for characterising software; McCabe and Halstead [11]. These metrics indicate the complexity of the software by assessing a number of properties, including number and depth of

branches. Whilst the complexity is clearly dependent on that of the specification, an indication can still be obtained of how hard the software will be to understand, reverse engineer, test etc.
Evaluation is performed on some representative samples of code produced by the AG.

Correctness:

A number of attributes, including correctness itself, are related to how well (well is defined as code meeting intent of specification) the AG transforms specification into code. The attributes include; all forms of testing, ability to qualify, and whether a safety argument could be generated. The correctness of transformation is evaluated both qualitatively by inspection and quantitatively by unit testing.

Coding Standard:

A key issue previously stated is that when considering verification attributes is that of the coding standard which the tool follows when converting the specification to code. The coding standard is important because of its effect on a number of attributes since it not only affects the output (i.e. code) but also the ability to relate the output to the input (i.e. specification). The three main issues related to the coding standard are; general suitability, specific suitability and configurability.
The *general suitability* evaluation is primarily quantitative by inspecting the code produced to determine whether it initially conforms to commonly held beliefs of what makes good software. One method for doing this is by using freely available tools such as *lint.* Lint can check whether software conforms with applicable standards and can reveal the use of unusual constructs that may be a source of subtle errors that a compiler may not be able to find. Normally, *lint* can be configured to evaluate the software to different degrees of rigour. The *specific suitability* evaluation considers adherence with more specific software standards such as SPARK Ada [7]. The *configurability* evaluation is performed qualitatively by experimentation.

Safety Argument:

The safety argument attribute brings together many of the other attributes. Other than cost it is probably the most important. Normally the safety argument is produced for a specific system since its aim is to justify that all reasonable steps have been taken to reduce the likelihood of a system's functional hazards (which can only be related to a specific system) to an acceptable level. However safety arguments, e.g. for use of modern processors [12], exist that deal with whether specific component does not introduce an additional hazard or make existing ones more likely.
It is not the intention to establish here whether a safety case could be produced for an AG. Instead, the aim is to perform qualitative assessment of whether the key forms of supporting evidence.

Summary:

Table 5 reviews the relationships between evaluation techniques proposed and attributes assessed.

Evaluation Technique	Attributes
Software Complexity	Traceability, Reviews, Control flow analysis All forms of testing, Hand tailoring, Other code, Cost of evidence, Maintainability
Correctness	Correctness, All forms of testing, Ability to qualify, Safety argument
Coding Standard	All forms of analysis, All forms of testing, Maintainability, Coding standard
Safety Argument	Safety argument, All forms of analysis, All forms of testing, Correctness, Completeness of verification, Assessment versus operational experience, Assess tool versus assess code

Table 5 - Relationship of Evaluation Techniques and Assessment Attributes

Evaluation Results

Simulink:

Simulink is a modelling and simulation tool for control systems. Simulink is referred to in its user guide as a "very high level language" a badge that is intended to reflect its superiority over conventional high level languages like C and Ada as a design tool, as "the manual process of transforming designs to code is largely eliminated". Simulink allows you to build up control systems very quickly using a "drag and drop" direct manipulation interface.
The "real time workshop" accessory for Simulink allows code generation to a variety of platforms, using either the Ada or C programming language. There are options to configure how the code is generated. The Target Language Compiler (TLC) transforms a Simulink file into C or Ada code. The TLC generates its code based on target files, which specify particular code for each block, and model-wide files, which specify the overall code style. TLC works like a text processor, using the target files and the model file to

generate ANSI C or Ada code. In addition, there are a number of different code optimisations under the control of the user, e.g. to reduce memory usage.

Software Complexity:

As part of our evaluation, we developed a simple averaging function in Simulink and used the embedded *mcc -x* compiler to generate C code. Then the *CCCC* code metrics package [13] was applied to the resulting code to determine code complexity.
The tool produces results such as those in Table 6 for a Simulink standard example that represents the flight controller for the longitudinal motion of a Grumman F14 jet.

Metric	Tag	Overall
Number of modules	NOM	2
Lines of Code	LOC	271
McCabe's Cyclomatic Number (measure of decision complexity)	MVG	33
Lines of Comment	COM	158
LOC/COM	L_C	1.715
MVG/COM	M_C	0.209

Table 6 - Metrics report for the F14 example

The first point to note is that the code for this more realistic system is less complex than might have been anticipated from applying CCCC to the first example. The complexity of the code generated does vary with the complexity of the model used as input. There is a degree of complexity associated with including the basic Simulink functions and structure required to provide a basic code outline, the additional complexity above this basic "template" follows a linear relationship with the complexity of the model used as input. This shows that the AG mainly performs a relatively simple syntactic conversion which means that all the structure and functionality including assumptions are captured in the model.

Correctness:

The code is constructed using templates: one for overall structure; one for each control system component; and finally another for translation to the target compiler. The translation from diagram to code is broken down into a number of well defined stages that may be assessed individually within a more restricted scope. This improves the capability to reason about each step within the limited scope of each template. For example, a safety argument for the code generator could be built up over the relevant templates demonstrating that at no point in the translation was a risk introduced to the code as a result of any mis-representation. This argument could be presented in a rigorous fashion for safety related code. For safety critical code a more formal approach is required.
A one-off formal verification of code automatically generated, with full optimisation, from a Simulink model has been performed by the Systems Assurance Group at QinetiQ. The Simulink model was for a flight control system featuring autopilot and autothrottle for an experimental aircraft. Just over 35,000 lines of SPARK Ada was generated and formally proven against a representation of the Simulink model in Z.
The technical approach has been described in [15] in addition a semi-automated refinement approach has been developed for a substantial verification [16]. This work indicates that the TLC approach could support automated verification and is the subject of current research.

Coding Standards:

The approach to constructing the code is deterministic and well defined and seems to deliver in the three main areas:

- *General Suitability* - There is a clear recipe to constructing the code from templates that is predictable. This means that the characteristics of the code in the templates can be inferred across the code it has been used to create - so long as an argument can be made that the template is only applied where it is appropriate. For example, if all the templates can be shown to have structured control flow, then all programs generated from these templates will also have structured control flow, as control flow between the templates is only ever sequential and addressed by the overall code structure template embedded into the code generator.
- *Specific Suitability* - The ability to comply with specific standards has been demonstrated by modifying the TLC files to generate SPARK Ada.
- *Configurability* - Provides a fine-grained approach to configuring the code generation process. Through a variety of options (Systems target files; Code Formats; S Functions) that allow fine-grained control over

how the code is constructed. However the options provide the opportunity to introduce new problems, such as new non-functional properties that could increase risk in the context of specific system environments.

Safety Argument:

Manual inspection of the software produced concluded that Simulink offers a predictable code generation system that supports verification of the code against the model. The way it constructs code is relatively straightforward and would be amenable to independent verification. This predictability is the tools main strength.

Problems are likely to stem from the wide variety of different options available for configuring the code. One approach to resolving such issues could be to agree a limited subset of the build options and the TLC files which is guaranteed to uphold basic implementation invariants that are considered critical to safe operation and enforce this subset rigorously.

Ilogix Statemate:

Statemate code generation, like Simulink, offers a predictable code generation facility based on the use of a template code structure and a comparatively simple code construction process. Code generation could be verified by validating the basic code structure used and then showing that each state is accurately represented by the relevant code fragment. In this way, the verification effort is likely to be a linear function of the models complexity. Difficulties would include the need to validate the library functions used (such as the "notify" routine used to broadcast system state data). These would need to be rigorously proved correct. However, this would be a one off argument for each version of the Statemate tool.

UML Scriptor

The weakness of tools such as Scriptor is that there is a front-end cost in defining the code generation profile in advance. Not only must this profile be complete, consistent and well defined, it must be validated as being appropriate for use on specific projects. Changes to the programming language, programming style, memory usage, timing requirements might also need to be reconciled in the original profile, and changes to this profile controlled to prevent the process becoming chaotic. Tools like Scriptor would allow visibility and development of the mappings between the model and the code, which would make it easier to reason about the tool's operation and hence its correctness. Knowledge of the tool's operation could support arguments that the use of the code generator has not introduced any new hazards or increased exposure to risks.

Conclusions

In this paper a number of evaluation attributes for AGs have been defined that can be used to assess COTS AGs to judge whether they are usable in the context of safety critical or safety related systems development. Usability is judged by the ability to certify the final product if an AG is used to generate some of the software within it and whether the use of an AG results in cost effective development compared to software generated by the traditional methods employed by programmers.

Using the method that has been derived, we have evaluated the AGs for Simulink, Statemate and Scriptor – only the results from the Simulink evaluation are presented here in full with the others summarised. This showed that the code generators were relatively predictable and simple in nature (largely based on a syntactic transformation). Key differences are the levels of configurability, and whilst relatively simple code resulted from the AGs there were notable differences in their relative complexities. The assessment framework highlights the areas where additional evidence would be required for use on a safety critical project.

References

[1] M. Whalen, M. Heimdhal, *On the Requirements of High-Integrity Code Generation*, Proceedings of the 4th High Assurance in Systems Engineering Workshop, 1999.

[2] C. O'Halloran, *Issues for the Automatic Generation of Safety Critical Software*, Proceedings of the 15th International Conference on Automated Software Engineering, 2000.

[3] S. Stepney, *High Integrity Compilation: A Case Study*, Prentice-Hall, 1993.

[4] J. Bowen, *Towards Verified Systems*, Real-Time Safety Critical Systems - volume 2, Elsevier, 1994.

[5] A. Pnuelli, M. Siegel and E. Singerman, *Translation Validation*, Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems, 1998.

[6] I. Bate and N. Audsley, *Architecture Trade-off Analysis and the Influence on Component Design*, Proceedings of the Workshop On Component-Based Software Engineering, 2002.
[7] J. Barnes, *High Integrity Ada: The SPARK Approach*, Addison-Wesley, 1997.
[8] *Guide for the Use of Ada Programming Language in High-Integrity Systems*, ISO/IEC PDTR 15952:1998, 1998.
[9] www.telelogic.com
[10] www.mathworks.com
[11] N. Fenton, S. Pfleeger, *Software metrics – A Rigorous and Practical Approach*, 2nd Edition, PWS Publishing Company, 1997.
[12] I Bate, P Conmy, T Kelly, J McDermid, *Use of Modern Processors in Safety-critical Applications*, The Computer Journal, 44 (6), pp. 531-543, 2001.
[13] http://cccc.sourceforge.net
[14] S. Burton, J. Clark, J. McDermid, Testing, *Proof and Automation. An Integrated Approach*, Proceedings of the 1st International Workshop of Automated Program Analysis, Testing and Verification, 2000.
[15] R. Arthan, P. Caseley, C. O'Halloran, A Smith, *Control laws in Z*, Proceedings of the International Conference on Formal Engineering Methods, pp. 169-176, 2000.
[16] C. O'Halloran, *Acceptance based assurance*, Proceedings of the 16th International Conference on Automated Software Engineering, 2001.
[17] I Bate, N Audsley, S Crook-Dawkins, Automatic Code Generation for Airborne Systems – the Next Generation of Software Productivity Tools, Proceedings of IEEE Aerospace Conference, 11-19, 2003.

Biography

N. Audsley, Senior Lecturer, Department of Computer Science, University of York, Heslington, York, YO10 5DD, U.K, telephone - +44 1904-432787, facsimile - +44 1904-432708, e-mail – neil.audsley@cs.york.ac.uk.

Dr Neil Audsley received a BSc in Computer Science and DPhil from the University of York in 1988 and 1993 respectively. His doctoral thesis considered the scheduling and timing analysis of safety-critical systems. He has published numerous technical papers. He is now a Senior Lecturer in the Department of Computer Science at the University of York, researching the design and implementation of real-time embedded systems.

I. Bate, Research Fellow, Department of Computer Science, University of York, Heslington, York, YO10 5DD, U.K, telephone - +44 1904-432786, facsimile - +44 1904-432708, e-mail – iain.bate@cs.york.ac.uk.

Dr Iain Bate has been a Researcher within the Real-Time Systems Research Group within the Department of Computer Science at the University of York since 1994. His doctoral research, completed in 1998, focused upon establishing and demonstrating an approach to scheduling and timing analysis for safety-critical systems. His research has mainly been in the real-time systems, aspects of safety-critical systems.

C. O'Halloran, A109, Woodward Building, QinetiQ, Malvern Technology Centre, WR14 3PS, UK, telephone - 01684 894320, facsimile - 01684 896113, email: cmohalloran@qinetiq.com

Dr Colin O'Halloran is a QinetiQ Fellow, a visiting Fellow at Kellog College, Oxford, and a visiting Professor in the Department of Computer Science at the University of York. He has developed verification techniques that his team have applied to a 35,000 line to a system consisting of flight control, autopilot and autothrottle. Currently these techniques are being developed for autocode and measured against traditional V&V techniques.