IMPROVING CERTIFICATION CAPABILITY THROUGH AUTOMATIC CODE GENERATION

Neil Audsley, Iain Bate, Steven Crook-Dawkins, John McDermid

Department of Computer Science, University of York, York, UK

{neil | iain.bate | steve | john.mcdermid}@cs.york.ac.uk

Abstract: Automatic code generation is a process of deriving programs directly from a design representation. Recent initiatives such as Model Driven Architectures mean they are becoming an essential component of software engineering and many commercial tools now provide this capability. Whilst these tools provide greater flexibility and responsiveness in design, they are also largely unqualified with respect to extant safety standards. This paper presents a summary of investigations into the issues in using autocode generators in critical systems, primarily avionic systems.

Key words: autocode generators, safety, MDA

1. INTRODUCTION

Recent initiatives such as Model Driven Architectures mean they are becoming an essential component of software engineering and many commercial tools now provide this capability. However these are largely unqualified with respect to the safety domain. The obvious response to the use of unqualified tools such as code generators in high integrity development is to perform extended verification, yet for this to be effective, much of the complexity of the coding process that the tool has automated would re-appear within the verification stage. Such verification may require a detailed knowledge of the design of the tool and this is often not available with commercial tools. More crucially, the costs of verification would be repeated for each instance of code production eroding many of the benefits

of automatic code generation. If a model of the automatic code generation process could be constructed, greater understanding of the process could be built up allowing arguments about safe translation to be constructed and evidence to be recovered. Given such a model, individual tools could be assessed on a one off basis for their ability to uphold the requirements of the model, or new tools could be developed to conform to the model. The contribution of this paper is to build up a framework for such models that would discharge the requirements for dependable code generation put forward by other authors:

In (1) Whalen and Heimdahl established five requirements for high integrity code generation, we will assess our model against these requirements:

1. Source and Target languages must be 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.

In this paper, the term Autocode refers to any piece of code generated from a tool rather than a hand coding process. The tools themselves are referred to as Autocode Generators or AGs.

2. OUTLINE OF APPROACH

There appears to be two ways to address the problem of arguing about the behaviour of an autocode generator:

1. Show that the AG itself can be verified to some definition of high integrity *across all instances* of its use as a one off argument, or
2. Verify the output of the AG *for each instance* of its use against a stable definition of performance.

The important difference between the two strategies is that the first would require an understanding of the internal structure of the AG, whereas the second would not. O'Halloran (2) argues that verifying an automated code generator is unlikely to be commercially viable.

Rather than attempting to formulate complex, fragile arguments that are directly related to individual, specific tool design or architecture, it would

make more sense to reason formally about the mapping from a design notation to the corresponding program code.

A set of mappings is used to argue about the behaviour of the AG, rather than attempting to argue about its internal structure. The rationale behind this approach is that a guarantee (or specification, definition, etc.) of a component's behaviour can be expressed at least an order of magnitude simpler than the implemented device. This approach would also provide a rigourous basis on which to discharge Whalen and Heimdahl's requirements for high integrity code generation.

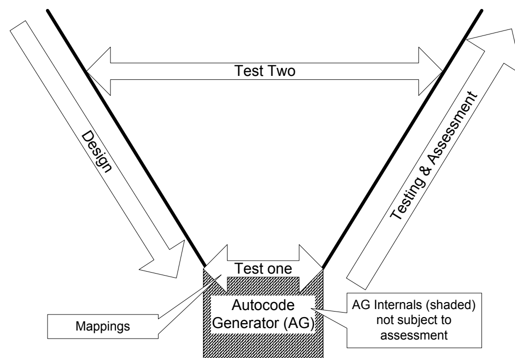

Figure 1. Isolating the Auto-code Generator

This mapping from design to code could be deployed through a two stage verification process. The first stage (Test one in Figure 1) is about the correctness of construction of the code by the AG as a refinement of the design. This test can be carried out by breaking down the input notation into basic components that map directly onto coding templates.

This approach also breaks down the proof into a set of arguments about each mapping, showing how the semantic meaning of the input construct is preserved in the corresponding code template. We believe this inductive or divide & conquer approach helps to alleviate O'Halloran's (2) concerns about the difficulty of verification through proof, by constraining each individual proof to only a single semantic concept.

It would be more cost effective to perform validation tests for safety requirements at a higher level of abstraction, as these tests can address performance and safety requirements directly (under test two). This separates the general problems of verifying of the AG from the specific problems of validating a given system against its requirements. This separation is important for (at least) three reasons:

1. Verification of the AG requires a different set of skills and tools to validation of the resulting code against performance and safety requirements.

2. Combining arguments about AG performance and System performance would make it impossible to disengage performance and safety claims from specific AG technologies. This would frustrate efforts to improve general capability for using AG tools, making AG use a project concern rather than a common concern across all developments.

3. Certification bodies will require evidence that the AG (and other similar development tools) have not introduced faults. This is in addition to a system level argument showing that overall risk is acceptable. The two issues are distinct, and arguments will be more compelling if they are addressed explicitly.

Considering the difference between safety and correctness reinforces these points. The concept of safety is concerned with risks of deploying a system within the context of specific environment (3). The mappings in Figure 1 only provide information about how the AG operates they do not present any claims that it is safe to use the AG or its output in context. Referring back to Figure 1, test one is about correctness; test two about safety.

It is not possible to make a **safety** argument for an AG out of context, as there is no way to gain a full understanding of the system hazards without this context. It would only be possible to verify the use of the AG against a common coding standard for a given design notation outside of a specific system context.

For AGs provided as COTS[1] tools by a third party supplier there may be limited information available to construct a set of mappings that define the coding standard. It may be possible to construct the mappings based on the anticipated behaviour of the AG then observe actual performance relative to these mappings. If the AG fails to uphold all the mappings, then the limitations of this AG in a specific context can be recorded and perhaps addressed elsewhere in the development process.

The offer made by some tool vendors of certification kits for the use of some AG tools may help in this regard. Such kits amount to a certificate from a standards-setting body showing conformance to specific standards and often permit access to specific evidence. However such kits provide little improvement in the capability of the development process to accommodate automatic code generation. As part of a study undertaken by Praxis and QinetiQ on COTS software, certificates for Real Time Operating Systems were described as usually insufficient due to the absence of any evidence from the design process (4). Rather than attempting to specify a

[1] COTS=Commercial off the Shelf - tools provided on a commercial basis not normally intended for safety critical development.

perfect AG, suitable for safety critical use, it is necessary to verify the performance of an AG in context.

Having defined our basic approach, the next section considers how feasible this approach is in the context of tools that must provide a useful service.

3. PROBLEMS WITH EXISTING TOOLS

The need to ensure a predictable process would motivate the use of mappings that are as simple and straightforward as possible - resulting in the use of a design language very similar to programming code. Yet such pseudo code would offer little to enable or encourage systems level safety analysis. Therefore any AG would need to trade off the need to generate correct code with the need to accommodate an appropriate design metaphor or language. This trade off is illustrated in Figure 2

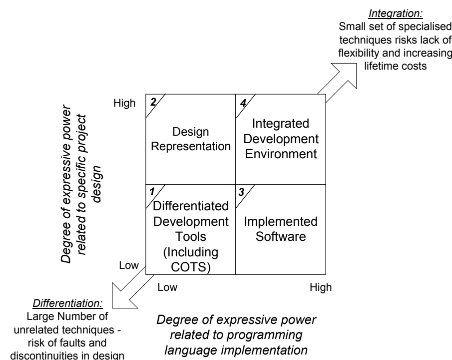

Figure 2. Trade off between different types of expressive power

There are four broad classes of trade off for the AG:

3.1 Differentiated Development Tools: Many of the tools are designed around a specific design methodology, such as UML or statecharts that are general modeling approaches. Whilst there is support for proofs of correctness and behavioural analysis through animation, these methodologies don't necessarily lend themselves to more investigative safety analysis techniques. They may require the support of additional tools to generate code, and are usually black box devices that are difficult to customize to specific requirements.

3.2 Design Representation: Within this group would be tools and techniques that are primarily concerned with modeling the project or system

to be implemented. Examples would include the use of HAZOPS[2] on piping diagrams in the chemical industry. These tools help to influence safe design, but provide little to guide code generation.

3.3 Implemented Software: These are tools are techniques primarily associated with supporting implementation of software. Tools such as software fault trees (SFTA) would fall into this category. They offer limited facilities for manipulation of the design, their focus being on the construction of the code itself.

3.4 Integrated Development Environments: These devices provide a total translation solution for a small range of applications, such as aircraft cockpit systems. If each application area used different tools and methodologies, then our ability to construct a safety argument across several systems would be compromised by the need for a different argument pattern for each area.

Whilst it would be possible to develop a process for automatic generation of code using any of these tool types, none would represent a general approach to provide sufficient design expression whilst generating verifiable code. This is because they tend to specialize in a particular approach or technology rather than addressing the whole problem of translating across the matrix from project and design concerns to implementation concerns.

Relating this back to Whalen and Heimdhal's original five requirements, the table below provides (at a very broad level) the suitability of each tool to high integrity code generation:

	R1	R2	R3	R4	R5	R6*
Differentiated Tools	😐	😐	☹	☹	😐	☺
Design Representations	☺	😐	😐	☹	☹	☹
Implemented software	☺	☺	☺	😐	😐	☹
Development Environment	😐	☹	😐	☹	😐	☺

Table 1: Broad assessment of suitability using Whalen & Heimdahl's requirements

** Additional requirement added - see point (2) below*

Where:

☺: This type of translator is ideally suited to discharge the requirement

😐: This type of translator could be specialized to discharge the requirement

☹: Discharging this requirement with this type of translator may not be feasible, either economically or technically.

The key points from this analysis are:

(1) Design representations may emphasize syntax and semantics of a design representation (R1), but economic viability may prevent rigorous

[2] HAZOPS – HAZard and OPerability Studies. This is a systematic method for assessing models against a number of anticipated failure modes, and was first described by Trevor Kletz (5)

analysis (R3, R4) and code quality (R5) would be a secondary concern for such tools

(2) Implemented software would (unsurprisingly) meet many of the requirements, but the rigour (R3,R4) of tools (such as compilers) that manipulate code remains difficult to reason about, and doesn't guarantee well structured code. The other obvious problem is that these tools offer little advantage because they do not support a design method to help derive code, and therefore offer little advantage over conventional technology. To address this, an additional requirement (R6) is suggested for AG's such that they must provide sufficient expressive power to make the translation useful.

(3) Finally, the development environment would perform many complex translations that would be difficult to reason about (R2). This additional complexity making rigourous testing (R4) infeasible.

The implication of this is that no single type of tool addresses all six requirements. A more general approach is required which takes on board all six requirements within the architecture of the AG. The next section provides discussion of the possible architectures that could be used, and the pros and cons of each.

4. REVIEW OF ARCHITECTURES

Three alternative approaches to a basic architecture for an AG have been put forward in (8). In Table 2 these approaches are identified and fitted into the general groups of tools proposed earlier.

Type of Mappings	**Type of Tool supported (from Figure 2)**
Black Box	Differentiated tools or development environment
Mapping Driven, Single Pass	Design Representations **or** Implemented Software
Mapping driven; Multiple Pass	Both design representation **and** implemented software

Table 2: Comparison between architectures and tool types

The black box AGs can only provide a specific solution to the translation and therefore would be restricted to differentiated tools or development environments, neither provides the insight required to formalize the translation rigorously as required by Whalen and Heimdhal (see table 1) and aren't adaptable to specific development requirements or coding standards.

The mapping driven, single pass (MDSP) AG breaks down the complexity of translation in one dimension by addressing the *breath* of the conversion process. With this white box approach it does not matter how many constructs are in the input or output languages – as each will have its

own mapping. However, tools based on this architecture cannot unpack complex (or deep) structures. The multiple pass architecture (MDMP) takes the next step, breaking down the *depth* as well as the *breadth* of the conversion process. The multiple passes allow the conversion of expressive power from project design concerns to implementation concerns to be controlled in a number of stages, each of which can be defined and verified. It therefore provides the only architecture to meet both the formal rigour required by Whalen and Heimdahl, whilst retaining the expressive power required for a useful AG tool.

Figure 3 illustrates the approach, showing how three passes (or, tiers) of mappings could be used to achieve translation, each set of mappings achieving a separate aspect of the process.

The different passes provide a way to combine the design representation tools with the implemented software tools, by reasoning about each tool as implementing as a separate set of mappings, which can be directly verified. This means that the translation problem is broken into meaningful steps instead attempting to describe the entire translation from design to code in one single, large, step.

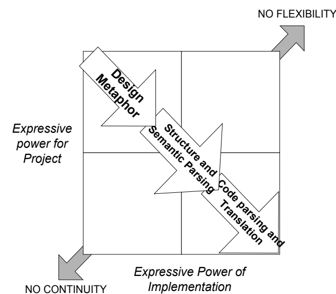

Figure 3. Controlling the change in emphasis from design focus to implementation focus

This builds a tool chain able to deliver on the six requirements without being compromised by the need for one tool to perform the whole job. This approach also has the benefit that the intermediate representation passed between each tool is a model of the system that can be stored in a standard recognized form, such as UML, or XML - preventing the need to lock in to specific tools or specific versions of those tools. One final point is that the application of mappings to refine the model from one stage to the next permits faults to be identified as soon as they occur, setting the code generator to a fail safe state that prevent anomalies propagating.

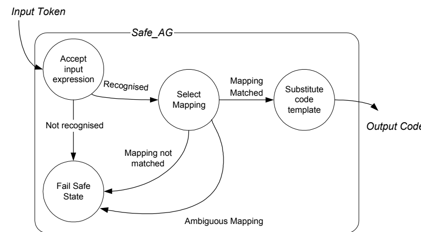

Figure 4. Simple fail safe approach to implementing mappings

This is a structured approach and is amenable to structured argument and/or proof. Using the Goal Structuring Notation or GSN (6) we have been able to argue about a simple AG. GSN allows arguments to be built up by systemically decomposing claims down to a level at which the lower level claims can be discharged by direct reference to evidence. This method of constructing arguments parallels directly the decomposition of the translation process into a set of mappings that can be verified directly. Note however, this is not a safety argument, as this can only be constructed in the specific deployment context. It is merely a argument that the AG has meet the requirements of a given coding standard defined by the tiers of mappings. Putting this another way, it is an argument that discharges test 1 in Figure 1, but only provides support for the broader safety argument required to discharge test 2. Other work performed by the authors has presented the arguments generated and considered how the resulting evidence needed can be generated (7).

5. CONCLUSIONS

The traditional arguments against the use of AGs in high integrity developments are mainly relevant to one specific type of AG, the black box, popularised by the use of COTS products with autocode facilities. We concur with Whalen and Heimdahl that rigorous arguments and formal definitions will be required in any dependable autocode technology.

A useful AG must have the ability to manage the shift in expressive power from design-centered tools to implementation tools. Design tools must have the flexibility to elicit system design issues, whilst the implementation tool must be a predictable model of a defined language or a specific platform. We identified four different types of tool that are available and discovered that no single tool architecture meets the dual requirements of facilitating rigorous proof whilst providing a translation powerful enough to be useful.

A mapping driven, multiple pass AG was suggested that systematically decomposes the translation process in both the *breadth of the language* through the use of mappings and *depth* through the use of multiple passes. The approach was recognized as being the most appropriate for use in critical systems. This decomposition approach mirrors very closely the approach taken to build up safety arguments, and makes the architecture amenable to rigorous analysis. Most crucially, it allows a code generation to be seen as the refinement of a model, using a tool chain which can be specified by mappings, and rigorously analysed and assessed.

6. REFERENCES

(1) Whalen M W, Heimdhal Mats P.E., On the Requirements of High-Integrity Code Generation, Proceedings of the Fourth High Assurance in Systems Engineering Workshop, Washington DC, November 1999

(2) O'Halloran C Issues for the automatic generation of safety critical software, Proceedings of the Fifteenth International IEEE Conference on Automated Software Engineering (ASE 2000), France, September 2000

(3) RTCA and EUROCAE, Software Considerations in Airborne Systems and Equipment Certification, Radio Technical Commission for Aeronautics RTCA DO178B/EUROCAE ED 12B, 1993

(4) Murray T, Simpson A COTS Software for High Integrity Applications, 36th Seminar of the Safety Critical Systems Club COTS & SOUP: Current thinking and Work In Progress , IEE London, April 5th 2001.

(5) Kletz, T. *Hazop and Hazan : Identifying and Assessing Process Industry Hazards*, Institute of Chemical Engineers, 3rd Edition, 1992.

(6) Kelly, T. P. Arguing Safety - A Systematic Approach to Managing Safety Cases, DPhil Thesis, Department of Computer Science, University of York, UK, YCST 99/05, September 1998

(7) Bate I, Audsley N, Crook-Dawkins S, Automatic Code Generation for Airborne Systems – the Next Generation of Software Productivity Tools, Proceedings of IEEE Aerospace Conference, 11-19, 2003.