Dependable and ubiquitous Autocode Generation

Neil Audsley, Iain Bate, Steven Crook-Dawkins
Real Time Systems Group, University of York, YORK, UK
{neil.audsley | iain.bate| steven.crook-dawkins}@cs.york.ac.uk

Abstract

"Automatic Code Generation" is a process of deriving programs directly from a design representation. Many commercial tools provide this capability. Whilst these tools provide greater flexibility and responsiveness in design, the market is technology-focussed and immature. No infrastructure or established theory exists which could be used to deploy the technology across large projects whilst upholding coding standards and safety requirements.

The objective of this paper is to develop a model or architecture for code generation that will be sufficiently well defined and unambiguous to support formal reasoning whilst also retaining sufficient expressive power to be useful. These models are based on statically defined mappings.

Introduction

The advantages of automatic code generation are very compelling- software for "free", given the presence of a design model. Vendors have recognised the potential for cost reduction and the ability to empower prototyping without recurrent system build costs. Against this, the market for tools to support "Autocode" is still at a chaotic stage. Tools differ in basic technology, platform support, design methodologies and programming language support to the extent that choosing a tool becomes a strategic issue. What little interoperability that does exist is clumsy and increases the potential for design faults to enter the system design. Few common principles exist to underpin such tools use across an organisation, let alone between organisations. With the increasing use of software on international and inter-organisational systems development projects, such as defence systems, a more dependable, mature process permitting greater interoperability will become essential if large scale projects are to benefit from the increased effectiveness autocode systems have to offer.

An important starting point for developing interoperability is the definition of standards. Whalen and Heimdhal[1] established five requirements for high integrity code generation

1. Source and target languages must have 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 the 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.

This is a formidable set of requirements and no tool vendor has yet discharged them. This is unsurprising given that only a small portion of these vendor's markets could be described as "high integrity", yet if Autocode systems are to offer a ubiquitous platform for software development then a common set of requirements will need to be established. This paper raises some of the issues we believe will be important in developing a model for autocode to meet these requirements and provide a foundation for improving the maturity of the autocode deployment processes. It also provides some suggestions for developing a more mature process for "autocode" tools that could allow standards to be defined to enable dependable and ubiquitous use of these technologies.

Outline of Approach

At first glance, two approaches to assessing Autocode appear viable:

- Trusted technology: show that the Autocode tool itself can be verified to some definition of "high integrity" across all instances of its use.
- Trusted process: Verify the output of the Autocode tool for each instance of its use against a stable definition of performance.

It seems very optimistic to expect that a tool can be validated for all present and future applications, [2]. Arguments based on specific tool technologies are likely to remain immature either due to the complexity and/or volatility of tool technologies or the emergence of new safety-related application domains.

Rather than attempting to formulate complex, fragile arguments that are directly related to individual, specific tool designs, a process based approach has been established that avoids the complexities of individual tool technologies and instead models the autocode tool as a set of translation mappings in the context of a conventional development process.

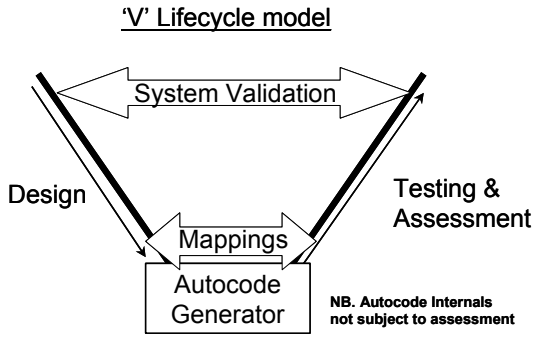

Figure 1: Using mappings to facilitate Autocode in a "V" Life cycle

The idea is to break down the translation performed by the autocode tool into a number of mappings, each of which describes a small part of the overall translation process from the design method to code. If each mapping translates from a single input design construct to the corresponding code fragment then it would be possible to verify each mapping is correct by showing that the semantics of the input design construct and the resulting code fragment are equivalent.

The key rationale behind this approach is that the mappings, as an abstraction of one part of the tools behaviour, will be less complex than the underlying technology and can be adopted as a standard for several different autocode technologies. This "inductive" or "divide and conquer" approach provides a structure to the argument for tool correctness. It can be deployed across several tools, even if the tools use different technologies. It provides the potential to develop a mature assessment process. It will be effective as it constrains each element of the argument to a single semantic concept for the design language chosen and avoids the costs associated with complex arguments.

Verification and Validation

For dependable or high integrity systems development, a safety argument must also be constructed. It would be more effective to separate general tool verification from specific system validation (see figure 1). We believe 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 autocode tool performance and system performance would make it impossible to disengage performance and safety claims from specific autocode technologies. This would frustrate efforts to improve general capability for using autocode tools and interoperability of those tools.
3. Certification bodies will require evidence that the Autocode tool (and similar development tools) have not introduced faults. This is in addition to and in support of 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 separately – especially as arguments for verified tools can be re-used reducing costs and improving maturity as these arguments are subjected to wider review.

These points illustrate that "dependability" applied to a tool is a much narrower scope than "dependability" at a system level. Whilst a tool can be shown to be dependable by correctly implementing a set of mappings, dependable use of that tool to build a high integrity system requires us

to argue about the specific context. This leads us to suggest that a mature autocode tool should not only show compliance to a specific coding standard (defined by the mappings) but also provide support for a design language on its input that is amenable to safety and dependability analysis. These two requirements are not necessarily mutually consistent.

Expressive Power: Design vs. Implementation

Tools, in general, tend to be good for one specific purpose. General tools, such as programming languages usually result in a compromise, which is difficult to manage. With the Autocode tool, we require a well-defined set of mappings to provide a coding standard, yet we also require that it support a high level system design analysis method on its input – potentially making the mappings more complex. Any autocode tool therefore makes a trade off with respect to expressive power – between the need to capture the design in an appropriate (and usually, imprecise) design language and the need to implement that design in a precise programming language.

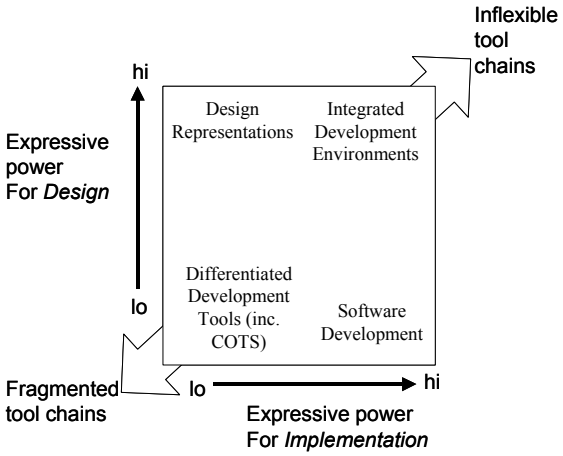

Figure 2: Trade off between design and implementation

In Figure 2, four common types of tool have been highlighted that currently exist. The "Design representation" tools address the need to construct a design and perform basic testing and walkthroughs of the concepts – they focus on design assessment, and ignore code construction. The "differentiated development tools" tend to be the design tools that vendors have added code generation features to based on a bespoke code generation technology. Most of these tools are limited in scope in that they tend to offer rather less than "100% code generation" – requiring extensive modification to the code output. Those that do offer full code generation usually do so within a very restricted context – such as a single design technique, such as state charts. Integrated development environments are suites of tools, usually connected through a common database, that support a development for a very restricted set of problems – whilst they score high on both axes, they are limited in scope and don't offer a general solution and focus on specific technologies. Finally, software development tools provide support for program construction and verification, but largely ignore design aspects – examples include SPARK annotations to Ada, and the "lint" analyser for C.

The implication of this is that no single type of tool (COTS, Design tools; Integrated Environments; or Software construction tools) will address our dual requirement for a precise coding standard and support for design analysis. Perhaps its unrealistic to expect both requirements to be addressed by a single tool – but since few of the tools are capable of interoperability, we have little choice. If tools did support interoperability, we could use separate tools to complete the process and navigate the optimum and most manageable route through the trade off:

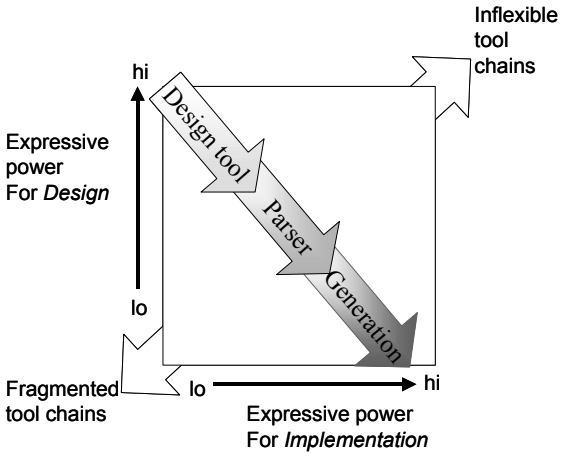

Figure 3: Connecting "available" tools together into an optimal tool chain

Breaking the process up in this way makes it more feasible that a mapping can be drafted to describe each tools operation, and use this to verify the tools operation. In addition, if the process consists of (for illustration) three tools each defined by a mapping, then each tool can be verified against its mapping, thus generating an "audit trail" of evidence about the process rather than assuming the process is a "black box". This enables faults to be identified closer to the source of the fault and improvements made. If each tool can be shown to conform to its mappings then ubiquitous use of these tools can be made across projects. Interoperability could also be supported as design teams could select their own design metaphor without breaking the ability to generate the code.

Implementing the process.

We know of no currently available tools that claim to conform to a published set of mappings. The closest being commercial compilers, but the language reference manuals (or "LRM"s) they "conform" to are still some way from the structured mappings we would envisage as necessary to discharge Whalen and Heimdhal's requirements. To illustrate this point, the LRM for the Ada programming language, despite being an ISO standard, generates hundreds of technical queries (or "Ada Issues") and these continue to be assessed by the Ada Rapporteur Group [3]. The C programming language has also prompted further clarification in the form of specific guidelines on its use in safety related systems [4,5]. This occurs because the LRMs are monolithic documents that describe an entire class of tools based on specific low-level technologies, rather than focussing on the specific problem of system model translation and refinement. To discharge Whalen and Heimdhal's requirements would require a different, more structured, approach. Reasoning over the specific semantic concepts to be translated provides a practical and mature framework on which to base the mappings. An illustration of is given in Figure 4 below showing the use of a number of library mappings, each backed up by a verification argument, capable of translating from UML to Ada.

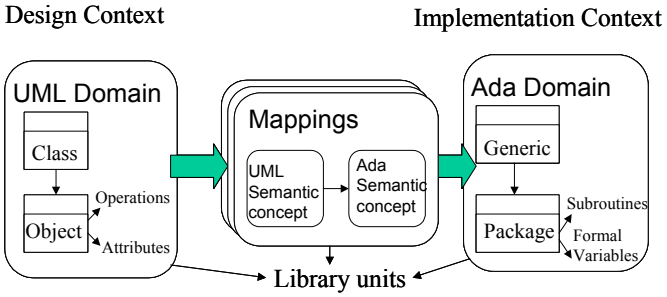

Figure 4: Schematic of the mappings approach for UML to Ada

The OMG's "model driven architecture" (or "MDA") [6] provides an excellent foundation for this work. The MDA is based on the need for greater interoperability across a number of diverging standards and technologies by defining a set of standards at the "meta" level – which means raising the level of abstraction to refocus on processes rather than specific technologies. MDA supports evolving standards from diverse range of applications, yet avoids the complexity associated with changes in

technology and the proliferation of middleware. Most importantly MDA ensures separation of concerns – most specifically separation of the logic of a specification from the detailed implementation. Standards are set for system description and translation that are independent of specific vendors or technologies– exactly what we have attempted to achieve within the narrower scope of autocode tools.

A demonstration autocode system has been developed to illustrate the feasibility of the approach, albeit within a very limited scope. The demonstration is based around the "eXtensible Mark up Language (XML). It uses a simple code generator written in the Perl programming language to instantiate a set of mappings from XML tags to C or Ada code.

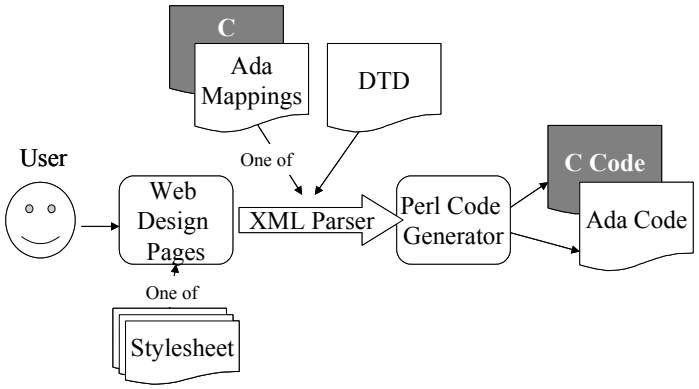

Figure 5: Schematic of the demonstration

Figure 5 shows the basic approach. The user enters the design using a number of web-based forms. This design is converted to an number of XML tags which are verified using the separately defined document type definition (or "DTD") which describes the correct structure for the "programs" described in XML. The validated XML is then sent to the Perl code generator. This code generator brings in a set of mappings from XML to either Ada or C. It is also possible to refine the design metaphor by setting up an XSL "stylesheet" – which uses simple template mappings to determine how the program design is to be rendered to the user, as a statechart for example. The demonstration can be viewed online at [7].

Rigorous Arguments

The inherent structure of this approach permits structured arguments to be put forward, and allows arguments to be updated and refined in–line with the tools themselves as the technology advances. A structured argument framework has been developed to support the research work at York. This framework was constructed using "Goal Structuring Notation" or "GSN"[8]. GSN allows the argument claims (shown as rectangles) to be broken down systematically and rigorously on the basis of explicit strategies (parallelograms), context (lozenges) and justifications or assumptions (ellipses). Figure 6 shows the top level of the argument structure, including the highest level claims. These claims would be broken down into lower level claims (not shown on the diagram) until the claims are sufficiently simple to be discharged by direct reference to evidence.

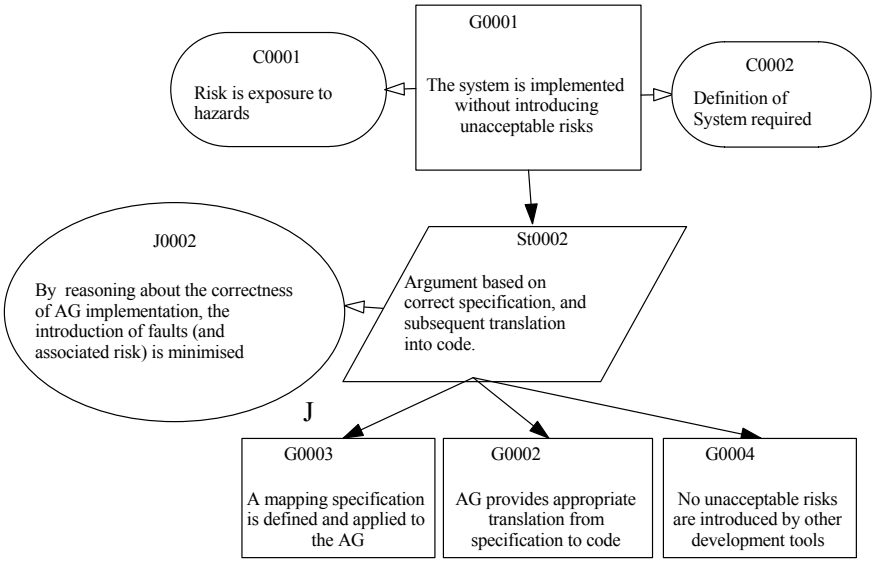

Figure 6: Top level of the structured argument in GSN

Summary

In a world of diverging technologies and increasingly dependence on large software systems the role of mature automatic code generation tools is increasingly important. However, the current immature state of the market, dominated by vendors keen to exploit new technologies provides a poor foundation for the ubiquitous use of autocode tools. We contend that process-driven arguments based on mappings and coding standards would help to analyse the situation and ensure the next generation of autocode tools are easier to reason about, and contribute to a mature and manageable process of translating from designs to code in a predictable and dependable manner. Ultimately this approach will help us to meet the requirements set down for high integrity autocode generation.

References

[1] Whalen MW , Hiemdahl Mat 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), Grenoble France, September 2000]

[3] http://pebbles.ocsystems.com/~acats/ais.html#AIS_Defect

[4] http://www.misra.org.uk/misra-c.htm

[5] Hatton, L. Safer C- Developing Software for High Integrity and Safety Critical Systems, McGraw-Hill International Series in Software Engineering, 1995.

[6] Soley R., "Model Driven Architecture" Object Management Group (OMG) White Paper, Draft 3.2, November 17, 2000 (www.omg.org/mda)

[7] http://www-users.cs.york.ac.uk/~steve/cgi-bin/loadfile.pl

[8] 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.