Automatic Code Generation for Airborne Systems[1]

Neil Audsley, Iain Bate, S. Crook-Dawkins
Department of Computer Science,
University of York,
Heslington, YORK, UK YO10 5DD
+44-1904-432765
{neil, ijb, steve}@cs.york.ac.uk

Abstract—The amount of software used on airborne platforms is increasing to unprecedented levels. With much larger amounts of software to design, control, and manage reasoning about overall systems performance becomes more difficult. This is a specific problem for Aerospace, as the use of newer technologies such as Integrated Modular Systems (IMS) and the need for high integrity systems further complicates the process and emphasizes the need to reason about system behaviour rather than specific software items.

This paper describes an approach developed within BAE SYSTEMS for a new generation of code generation tools that structure the code generation process to allow arguments to be made about the integrity of the code delivered. In addition, the approach breaks the development process down into different areas of concern. This allows any one aspect of the generation process to be reasoned about in isolation from the others, helping to broaden the scope of code generation without compromising integrity – an invaluable asset in the move towards more integrated aerospace systems.

TABLE OF CONTENTS

1. INTRODUCTION 1
2. PROBLEMS WITH EXISTING TOOLS 1
3. OUTLINE OF APPROACH 2
4. RIGOROUS ARGUMENTS 5
5. UML TO ADA FOR AIRBORNE SYSTEMS 5
6. IMPROVED PRODUCTIVITY 9
7. EMERGING TOOLS 10
8. CONCLUSIONS 10
REFERENCES 10

1. INTRODUCTION

Automatic code generation (or "autocode") is a technology for generating software from design analysis models with little, if any, human intervention. These tools, *if well conceived*, deliver a predictable, consistent and repeatable process and represent a step change in productivity for non-safety critical or federated systems. With the move towards greater integration of safety or mission critical systems on airborne platforms there is an increasing need for a dependable autocode technology capable of deployment in high integrity systems. If such technologies were independent of specific systems modeling approaches, this would empower developers to choose the most appropriate and semantically rich modeling language for the specific application system, and then apply a standard code generation application or template to deliver a representation of that system in software.

Whalen and Heimdhal [1] established a benchmark set of 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 maintain 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. *The generated code must be well structured, well documented, and easily traceable to the original specification.*

These requirements cannot be discharged by commercially available autocode systems that the authors know of. This prompted further investigation into the emerging techniques and technologies that would support the increasing business requirement in the UK Aerospace industry for dependable autocode systems.

2. PROBLEMS WITH EXISTING TOOLS

Automatic code generation is not a new concept; many tool vendors have offered the facility to generate code from design representations for many years. Such tools provide prototypes to allow further systems testing and refinement, however they were not intended to replace the existing processes of the software engineering discipline. The main reason for this is that the internal mechanisms these tools use to generate the code don't enforce any specific coding standards or structured techniques. As with any commercial

[1] Paper No. 1028 : 0-7803-7651-X/03/$17.00 © 2003 IEEE

tool, the quality of the output is largely a matter of trust until such time as the tool has been extensively tested. They are "black boxes". As the size and scope of the systems developed increases, reliance on testing is becoming less effective in comparison to "correctness by construction" approaches. If the tool is to be deployed in an almost infinite number of design situations – there may be very little read-across from one use of the tool to another. Each system the tool is used on will be different each will trigger a different set of paths through the code generator utility, the impossibility of 100% path coverage testing has long been accepted.

Further problems stem from the complexity of the tools. To be acceptable to broader commercial markets they have to be easy to use and support a specific development approach or technique. These techniques are often graphical (statecharts, entity relationship diagrams etc) and require complex editors and graphical user interfaces which add considerably to tool complexity. As the tool is a "black box" there is no way to isolate the code generator from these additional complexities reducing further our capability to rigorously test the tool. This integration raises another difficulty.

Finally, each tool tends to specialize in a specific technique that may cover only a small proportion of the overall systems engineering process – a statechart is not the ideal representation for hazard analysis for example. This introduces a co-ordination problem as a number of tools would need to be deployed simultaneously in the process to support all the different concerns associated with systems engineering. If code generation tools mandate the use of specific modeling techniques, this reduces the flexibility to use a modeling language appropriate to the development context. Ideally, the modeling approach and the code generation technology would be supported separately, but linked by an intermediate representation accepted by many tools vendors.

To summarise, we suggest there are (at least) three areas of concern for the use of current commercial autocode technology

1. Lack of support for rigorous testing / proof of properties and dependability
2. Additional complexity required in these tools to support graphical user interfaces and other requirements of a broad commercial market
3. Focus on specific technologies / methodologies that address only a small number of systems engineering issues and may not have the semantic richness to capture specific systems issues or problems in context

These are the issues stopping commercial tools reaching a platform to support Whalen and Heimdahl's requirements [1].

As a result, commercial autocode tools rarely provide a predictable, validated process for generating trusted code for use in dependable systems. Our requirement is for a tool sufficiently well structured to be verified, yet capable of encompassing a range of systems engineering issues.

3. 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, a point already made by O'Halloran [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 application areas that promote interest in a broader range of tools rather than deeper understanding of existing ones.

Rather than attempting to formulate complex, fragile arguments that are directly related to individual 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 such as the development lifecycle (or "V") model, see Figure 1

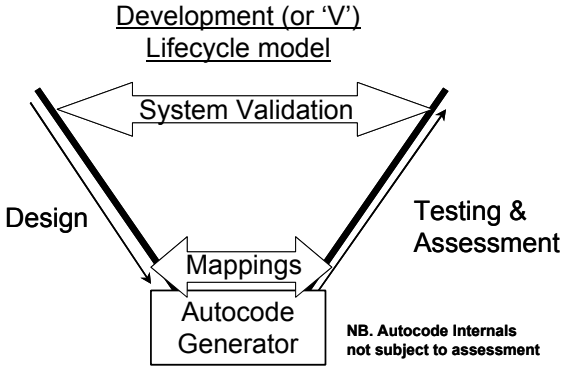

Figure 1: Using mappings to facilitate Autocode in a "V" Lifecycle

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 can be reasoned about independently and include only those

details required to define the translation to code. None of the complexities associated with graphical user interfaces or specific implementation platforms need be included. Such mappings could be adopted as a coding 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.

This approach helps to provide answers to the first two areas of concern identified at the end of section 2, but we have not addressed the need to address a range of systems engineering issues. In particular, by focusing on individual mappings, the expressive power of the translation as a whole has been ignored.

Expressive Power: Design vs. Implementation

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 precisely as within a programming language. The nature of this trade off is illustrated in Figure 2 below

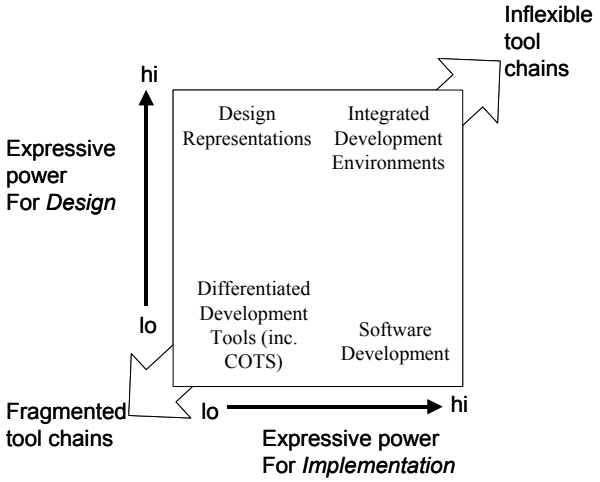

Figure 2: Trade off between design and implementation

In Figure 2, four common types of tool have been highlighted that currently exist. The details of each of these tools is less important than the idea that providing expressive power to reason about the design is not necessarily consistent with the need to express detailed and precise information during the implementation. Attempts to maximize both often result in a tool that is not only complex (and therefore difficult to reason about) but also inflexible. There is a limit to the number of design artifacts that can be accommodated whilst also providing a precise mapping of that artifact into code. High expressive power on both axes is likely to reduce the scope of the tool as the semantic richness of the tool becomes focused on a smaller set of systems problems. At the other extreme commercial off the shelf ("COTS") tools might rate highly on one or other axis, but rarely on both – resulting in a fragmented tool chain as more tools must be introduced to provide additional design or programming facilities. Our experience of currently available tools validates this – with all the current generation of tools falling into one or other of the four types we labeled on Figure 2.

Perhaps it is unrealistic to expect both design and implementation requirements to be addressed by a single tool – but since few of the current generation of 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 – this is illustrated in Figure **3**

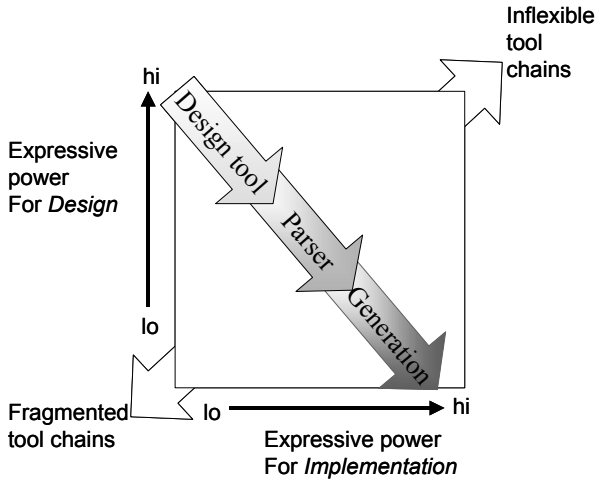

Figure 3: An optimal tool chain – avoiding the danger zones

Breaking the process up in this way makes it more feasible that a mapping can be drafted to describe each tool's operation as it resolves a smaller part of the overall translation from design to implementation. This is a crucial point as it enables the use of less complex mappings. There are two main reasons why we believe this to be the case:

- Within the context of each tool, the input and output languages will be "closer" on the design / implementation trade off and therefore are more likely to share underlying semantic concepts
- Within the context of the overall translation process, the scope of each individual tool is reduced. The output of each tool can be subjected to rigorous testing and locate faults or resolve ambiguities that a single monolithic tool would have to address within its internal design specification.

As a result, verifying the tool's operation will be more straightforward, less error prone, and feasible within realistic (i.e. project) timescales. In addition, if the process consists of (for illustration) three tools each defined by a

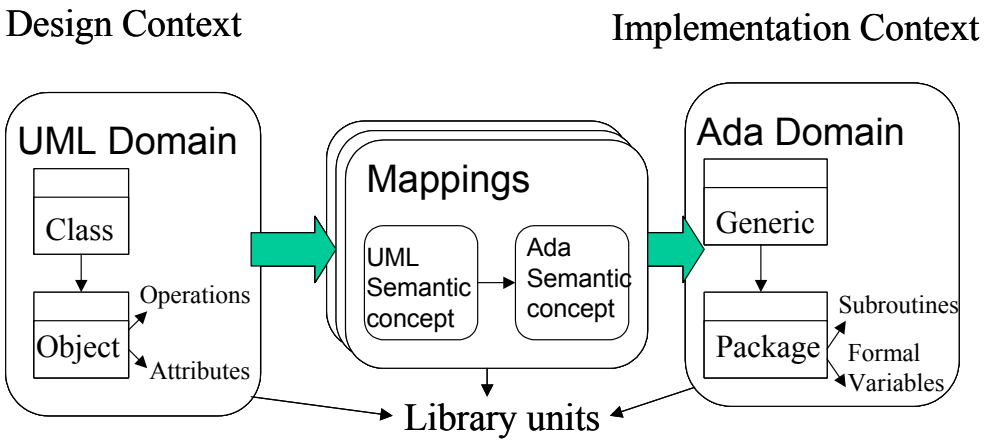

Figure 4: Schematic of the mappings approach for UML to Ada

mapping, then each tool can be verified against its mapping, thus generating an "audit trail" of evidence about the process.

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 through standardized intermediate representations allowing design teams to select their own design metaphor without compromising the quality of the code generation tools. This could be considered an "integrated, modular process".

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. For example, 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 focusing 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 the Unified Modeling Language (UML) to Ada.

The Object Managements Group's (OMG) "model driven architecture" (or "MDA") [6] provides a 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 – specifically separation of the logic of a system 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.

Simple Demonstration

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.

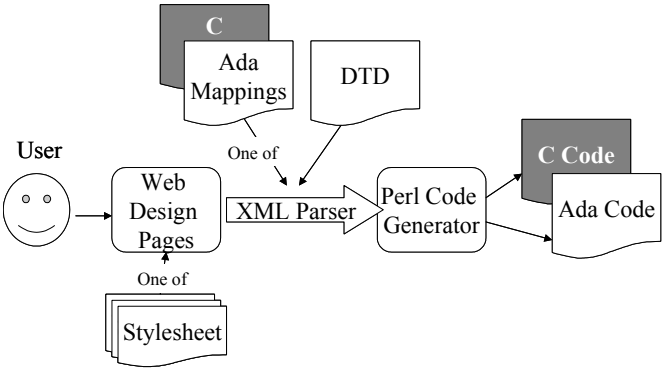

Figure 5: Schematic of the demonstration

Figure 5 shows the basic approach. The user enters the design using a number of web-based forms. This design is converted to a number of XML tags that are verified using the separately defined document type definition (or "DTD") that describes the correct structure for the "programs" described in XML (it represents a metamodel for program design). 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].

This demonstration system is not compliant to the Object Management Group's MDA specification, but is intended to illustrate the approach based on very simple technologies.

4. RIGOROUS ARGUMENTS

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 autocode generator 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.

The inherent structure of the mappings approach permits structured arguments to be put forward for tools, 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 a fragment of the argument structure for autocode generation in GSN. The highest-level claims are systematically decomposed to lower level claims. The implication of this decomposition is that a goal is not discharged until all the goals beneath it have been discharged. This means that less concrete, broader goals are broken down into more concrete and tangible sub-ordinate goals. Figure 6 shows one complete "audit trail" from the top-level argument down to more precise reasoning about individual mappings.

Whilst it may be possible to refine arguments to a point that they provide formal proof that a specific tool is acceptable for use in high integrity development. The more important point is that it provides a way to organize and present available evidence in a more compelling way as part of an argument. These arguments would then be used to address the requirements of specific safety standards, such as UK Defence Standards, which focus on an audit trail of evidence to support general safety and dependability claims. The structured argument is simply a tool that can be used to support and reflect the structured mappings we put forward for autocode generations. However, the ability to clear or certify a tool for use in high integrity systems will depend on to define a set of mappings, backed up with evidence of their correctness – we aren't suggesting you can short-circuit this by drawing boxes. The structured argument will provide a more concrete foundation to reason about the mappings and the associated evidence but consideration needs to be given to the type and structure of the mappings best able to support specific application domains.

5. UML TO ADA FOR AIRBORNE SYSTEMS

UML is becoming a ubiquitous standard for system modeling. A key goal of UML is that it is intended to provide a general systems modeling capability [10] as opposed to a narrow focus on a specific techniques. As a result, it is appropriate to consider the implications of a translator from UML to Ada to support airborne systems. Defining these mappings is complicated by the lack of formal semantics for UML. Our approach has been to define mappings and resolve ambiguities in the notation and record the implied interpretation unambiguously within the mappings. This is a practical approach and permits a degree of common understanding across a project. We would not claim to have defined a universal semantics for UML in these mappings however.

Our choice to translate to the Ada programming language is based on the use of Ada in the UK and Europe as part of the implementation strategy for dependable or high integrity systems. The process described here could also be used, with suitable code generation mappings defined, to generate other languages, such as C, Java etc.

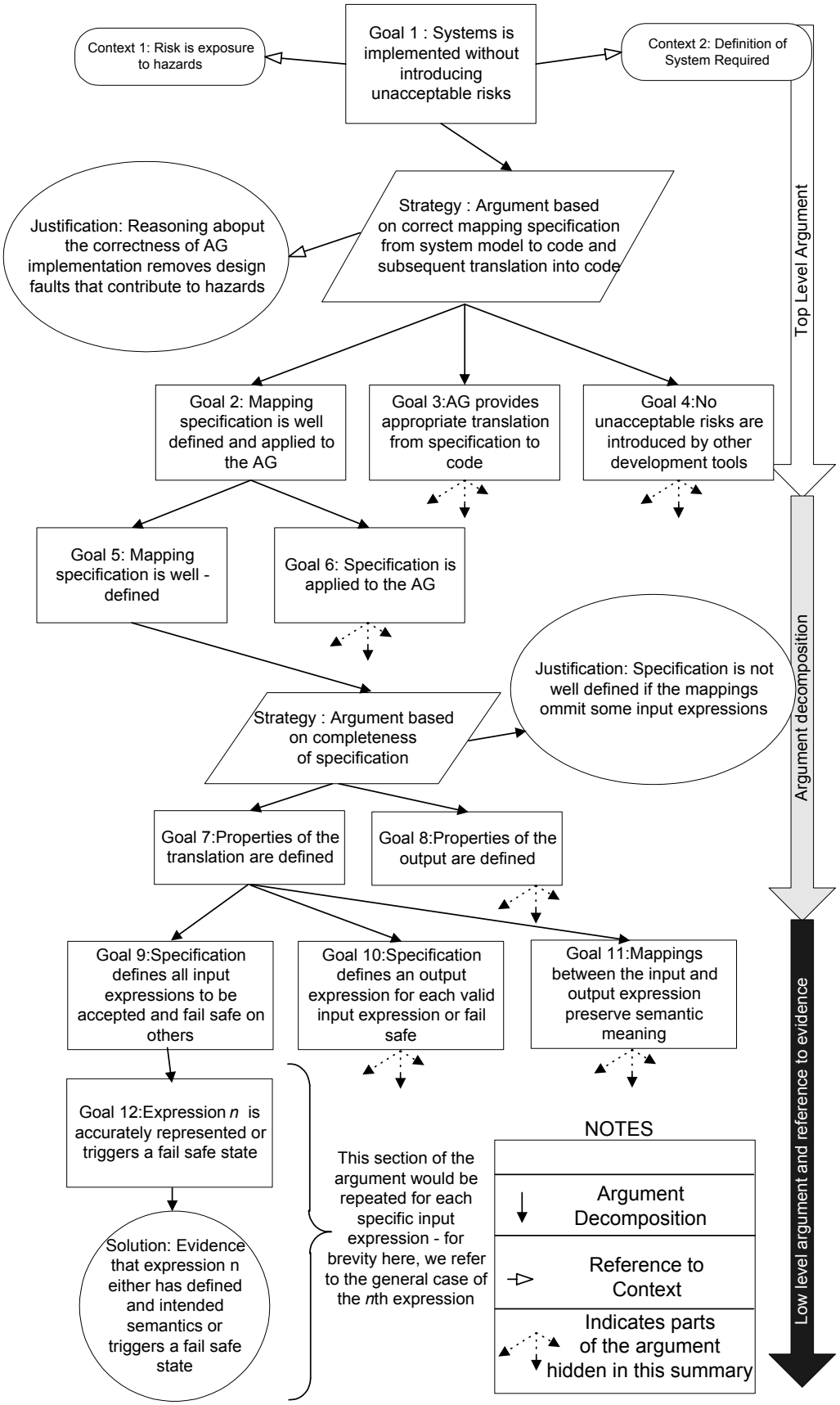

Figure 6: Fragment of the safety argument

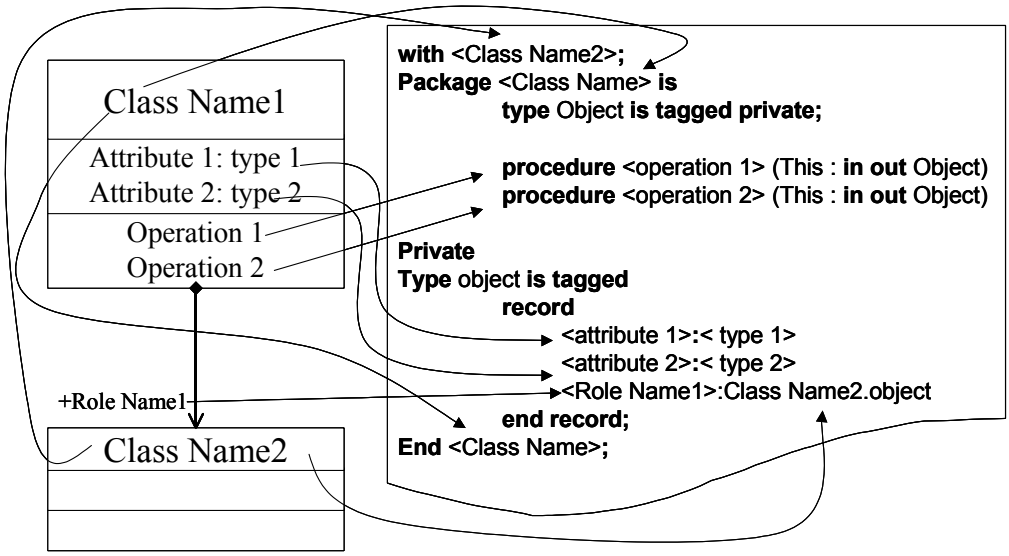

Figure 7: Dependencies using a direct mapping from UML to Ada95

As a starting point, we adopted the basic UML to Ada mapping described in [9]. These mappings focus on translating a UML class diagram into Ada 95. The mappings used define direct correspondences between the semantic units of UML to Ada. Whilst the mappings are well constructed, they would lead to complex implementations. For example, the UML "class" type is mapped to a tagged type in Ada. Figure 7 illustrates the dependencies that would need to be preserved to make this transition successful for a single UML class. The point is that the dependencies are complex to instantiate and manage.

The complexity of the process makes it difficult to reason about the correct construction of Ada code. For example, using the template shown in Figure 7, before a class can be translated, every class with an association to that class must be recognized and data from those associated classes used to populate the template. Whilst this is logically feasible, in practice this process of code generation will be inefficient and the resulting code hard to analyze. Also, the direct mapping shown requires that the structure of the UML model be imposed onto the Ada code – however, as we're already established – structures suited to design analysis aren't necessarily suited to implementation.

Need for a meta-model

To resolve this issue there is a need to disengage the design structure from the implementation structure. This can be achieved through the use of an intermediary meta-model – a simple example is shown in Figure 8. The intent of the meta-model is to describe the modeling approach using a standard definition. This allows each system model to be regarded as a single instantiation of this meta-model. If systems can be represented as instances of a standard meta-model, then the code generation program need only parse this one meta-model rather than reconstructing an internal representation from an infinite variety of different systems model structures – significantly reducing the complexity of the mappings and any rigorous arguments about them. Figure 8 shows a simple meta-model for class diagrams in the UML. The rectangles containing ***bold italic*** text are not part of the meta-model, but are used to illustrate the specific instances of the elements the meta-model needs to describe

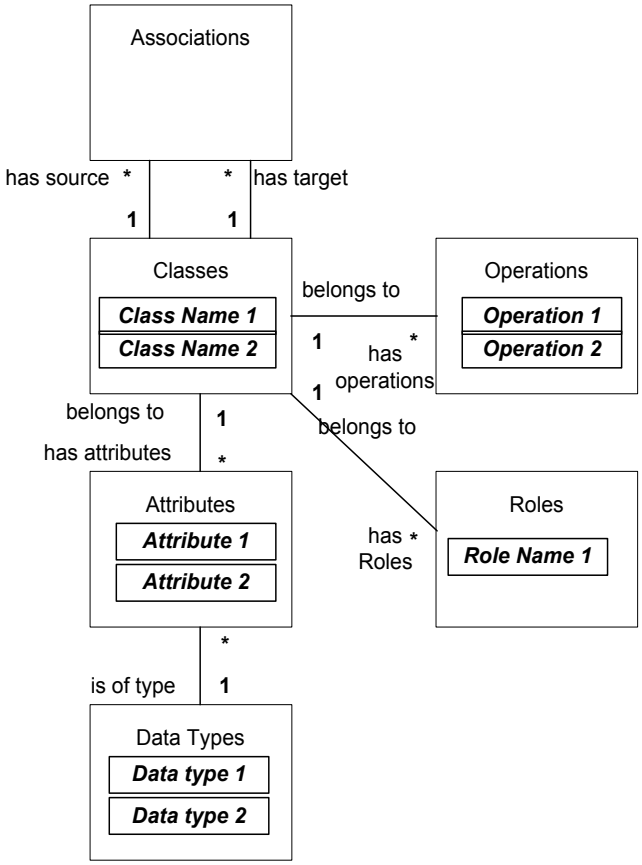

Figure 8: A simple meta-model (with specific instances required to describe the class model of Figure 7 shown in bold – these are not part of the meta-model itself)

the class shown on the left hand of Figure 7

The importance of defining this meta-model is that the code generation no longer has to contain a definition of the modeling technique and implement mappings to both deconstruct this model and rebuild the code model. Instead, the code generation algorithm can simply interrogate a standard meta-model and build up the code depending on the specific instances within that model. To illustrate, Box 1 below shows a basic code template showing the basic form of a mapping that would allow Ada to be constructed from any set of UML classes conforming to the basic meta-model shown in Figure 8.

```
For every <class >
  For every <class.hassource>
 -- include a with clause : (with class.hassource.target);
  end loop;
   -- header clause (Package <class.classname>) is
   -- define tagged object type
      for every <class.has operations>
        -- define  procedures to handle operations
      end loop;
      -- define private part (extend the tagged object type)
      for every <class.hasoperations>
        -- define a record entry
           (class.has attribute.attributename)
        -- : (class.has attribute.isoftype.datatype)
     end loop;
end loop;
```

Box 1: Basic Pseudo code for a mapping to construct Ada tagged types from UML class diagrams

Of course, this template would require considerable additional development prior to deployment. The key element is that the code can now be mapped to the template in a linear fashion – as the meta-model provides a way to structure information about the design representation and the dependencies between semantic units, rather than the design itself. The pseudo code above represents the mapping and it consists of two primary elements:

- A control structure to re-construct semantic meaning by iterating over the elements of the meta-model that share semantic dependencies
- A set of syntactic elements which make up the program statements

These two elements serve different purposes. The first defines the basic algorithm for constructing the semantic units into program code; the second defines the lexicon of the specific language generated.

The meta-model and the template used to generate code from it are both intended to be stable structures that can be analyzed, tested, even formally proved once, then deployed several times. Since the concerns of "define the modeling language" and "define a translation from model to code" have been separated, changes to either the modeling approach or the code generation policy can be implemented by updating either the meta-model or the code template respectively and re-validating that component. In comparison, using a bespoke and direct code generation policy (similar to that shown in Figure 7) would involve a more complex change and updating process. Not only is this more likely to introduce design faults into the generator, but it also means revisiting the trade off of design clarity against implementation precision for every change – making preservation of a clear, consistent policy on either more difficult to sustain

This approach is especially useful for airborne systems because it supports the need for rigorous testing and assessment required for high assurance software using the argument structures shown earlier. Three very specific concerns (scope, definition of code mappings, and safety,) have been supported by three distinct viewpoints. The scope of the modeling approach is supported by a metamodel, definition of code generation by a template; safety analysis by a structured safety argument. This separation of concerns is a key aspect of Model Driven Architectures and helps to support more integrated modular avionics systems.

Integrated Modular Avionics (IMA)

Traditionally systems safety assessment on critical platforms, such as airborne systems has relied on redundancy to mitigate the effects of failure in safety-critical applications. IMA systems, which aim to exploit greater interrelationships between systems, challenge this. For example, a safety critical system such as the flight control system may depend on inputs from the navigation system traditionally viewed as a mission critical system.

The separation of concerns supported by MDA offers several important assets to resolve this:

1. Focus on systems behaviour at a model, rather than code level, helping to integrate different applications into common system level services
2. Separates system models from implementation detail allowing applications to be deployed on different computing elements, or cabinets. Even permitting reconfiguration to ensure critical services continue to be supported in the presence of hardware failures [11]
3. Separation of concerns: using the appropriate meta-model to reason about one aspect of the systems behaviour without requiring that aspect to be physically and functionally isolated from other aspects of the system. In this paper we have provided aspects to allow reasoning about safety and code implementation, related to but separate from the UML model that describes the general system model. This not only makes it easier to

reason about aspects of system behaviour – it also helps to develop understanding of these different aspects to improve process capability and maturity.

By moving away from the traditional "evolutionary" lifecycles where specific issues (such as, for example, coding practice, or choice of implementation language) could only be addressed at specific points in the process, model driven approaches allow new technologies to be exploited more effectively. Technologies can be exploited where they are relevant and add value rather than requiring all aspects of the project to first assess the implications for their own aspect of the project and checking for dependencies.

The system as a whole (i.e. the final deliverable that represents the combination of all different views of the system including analysis model, design model, safety argument, code and test results) would need to be shown to be consistent. The safety argument provides an integrating role to demonstrate the inherent risk of systems deployment. If the safety argument cannot be discharged, then new evidence or alterations to the design may be required to reduce specific risks. As each key element of design development is now treated as a discipline in its own right – rather than being subservient to the needs and pressures of a specific project - there is an improved chance that a resolution can be found. For example, the use of exceptions handlers in Ada causes problems in gaining evidence about the code delivered because they introduce non-determinism that cannot be comprehensively tested. To address this a program template can be provided which removes exceptions from the programs generated – using instead a mechanism that is more predictable and easier to test.

Incremental certification and Tagging

One refinement might be to allow the code generation template to be altered depending one some user defined variable. For example, it may be required to generate code with a smaller memory footprint or optimize the code to reduce run-time.

Using a code generation template such as the one defined earlier in this paper, it would be comparatively easy to introduce this concept as it would simply require a refinement to one template rather than requiring an entire programming team to change their practice and re-work their code.

For example, exceptions in Ada might only need to be omitted in safety critical elements of the code, but might be acceptable for use in non-safety critical functions. If areas of the model can be tagged as being either safety critical or non-safety critical then the code generator, on reading the tag, could use a template that excludes the use of exceptions if the tag identified the section as being a safety critical function. This allows consistent choices to be made across the system model about specific safety or implementation requirements.

By making a code templates more specific, constructing different templates to address different coding problems and being clear about the instances in which each template is invoked, then the implications of a change to the code generator are more predictable and only those templates that are changed need be re-assessed. This is known as "incremental certification" and is considered a major benefit for implementing integrated modular systems. Maintaining the safety argument as a separate independent artifact provides greater potential to identify inconsistencies and faults across templates - instead of adopting independence in the product; independence will instead play a similar role in the process. This allows the compromises and trade offs to be worked out during product design rather than in the product itself.

6. IMPROVED PRODUCTIVITY

The amount of software on airborne platforms is set to reach a level that challenges our ability to deliver the program code itself, let alone support this code with safety arguments and maintain it over a life cycle of an operational aircraft. Add to this the need for more integrated systems and the current approach of hand-crafting software code quickly becomes untenable.

Automatic code generation provides an improvement in software productivity of similar magnitude to the introduction of compilers and the "high level" programming languages like Ada, C and so on. High-level languages where introduced when systems became sufficiently large and complex to make assembly code largely inefficient as a way to construct these systems. Assembly code did not allow the programmer to structure the detailed implementation in a way that was amenable to analysis. This is a direct parallel with the situation that is emerging now with high level programming languages. As the applications we wish to construct are no longer isolated in terms of technology, requirements, modeling methods and safety standards. It is now time to consider moving towards a new way of building systems that recognizes the importance of developing the common responses to these common issues and framing them in structures that support improvements and working towards greater maturity in development processes.

Auto-code is an important facilitator of this approach because it addresses one of the most volatile and therefore costly aspects, software management and construction. If dependable autocode systems can be delivered, structured around a set of stable and consistent domains, such as modeling technique, coding standard and safety arguments this would improve software productivity that is vital for the next generation of integrated airborne systems.

7. EMERGING TOOLS & FUTURE WORK

As a result of interest in Model Driven Architectures research within the OMG and increasing demand for greater configurability of code generation tools, there is interest from tool vendors to develop tools to support mappings and model driven approaches. To allow us to develop this work on a firmer industrial level we are currently negotiating a joint project with a tool vendor to implement a full set of UML to Ada mappings suitable for deployment on industrial projects.

8. CONCLUSIONS

There is an increasing business need to improve software productivity and place code generation on a more mature and efficient foundation. The technology exists to generate code from design notations but the market for these tools remains fragmented and does not support the need for rigorous arguments required for dependability, high integrity or safety critical systems such as airborne flight control systems.

In response to this, an approach is put forward for translating from expressive design notations to specific code implementations that treats each element of design (model approach, coding standard and safety argument) as a separate stakeholder in systems development. By defining meta-models to describe modeling languages and templates to map from these to specific implementations it is possible to place code generation on a more mature foundation – one that would support rigorous arguments.

ACKNOWLEDGEMENTS

The authors are grateful for the support of BAE SYSTEMS who funded the research work described in this paper under the "Systems Integration Consortium" project.

REFERENCES

[1] Whalen M W, Heimdahl Mats P.E. "On the requirements of High-Integrity Code Generation", Proceedings of the 4th 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 Conference on Automated Software Engineering (ASE 2000), IEEE Computer Society

[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.

[9] Taylor B, Karlsen E W, "Mapping UML to Ada", A Strohmeier and D Craeynest (Eds.): Ada Europe 2001, LNCS 2043, pp359-370, 2001.

[10] Rumbaugh, Jacodson, Booch The Unified Modeling Language Reference Manual, Addison Wesley Longmann, 1999

[11] Blackwell N, Dawkins S, Leinster-Evans S "Developing Safety Cases for Integrated Systems", IEEE Aerospace Conference, 1999.

Neil Audsley *is a senior lecturer in the Dept. of Computer Science of the University of York (UK).. Neil is a member of the Real-Time Systems Research Group (RTSG) with research interests in a number of aspects of real-time systems, including kernels / operating systems, scheduling algorithms; communications; programming languages for real-time systems and use of reconfigurable hardware devices.*

Iain Bate *has been working as a Research Associate since 1994. From 1994 until 1998, Iain worked within the Rolls-Royce University Technology Centre on fixed priority scheduling and high integrity systems. Since the beginning of the 1999, Iain has been working in the British Aerospace Dependable Computing Systems Centre looking at how advanced processors can be safely and effectively used in safety critical systems and the design and certification of Integrated Modular Avionics (IMA). Iain is a member of The Institute of Electrical Engineers and chairs their Informatics division's Software Engineering committee. He holds Chartered Engineer status. Iain has also recently completed a DPhil titled "Scheduling and Timing Analysis of Safety Critical Hard real-time Systems".*

Steven Crook-Dawkins *graduated from the University of York in 1993 with a Masters Degree in Computer Science. Since 1995 Steve has worked as a research associate looking at systems safety and hazard analysis. In this role he has acted as a consultant to BAE SYSTEMS across many*

systems development programs advising on safety assessment techniques and certification to UK safety standards. Since 2000, Steven has been working on the development of dependable autocode technology within the Real Time Systems Research Group. Steven is a member of both the British Computing Society and the Institute of Electrical Engineers.