Proceedings of the International Multiconference on
Computer Science and Information Technology, pp. 303 – 311

ISSN 1896-7094

Automated Code Generation for Safety-Related Applications: A Case Study

David P. Gluch, Andrew J. Kornecki

Embry Riddle Aeronautical University (ERAU)
Daytona Beach, FL 32114, USA
{gluchd, kornecka} @erau.edu

Abstract. This paper addresses issues relating to the suitability of using automated code generation (ACG) technologies for the development of real-time, safety-critical systems. This research explored the characteristics of model-based software development methodologies and the automated code generation tools that support them. Specifically, data related to the engineering challenges, skills, and effort associated with ACG practices and technologies were collected as part of a case study. Characteristics such as the generated code's organization, size, readability, traceability to model, real-time constructs, and exception handling were identified. In addition, the case study involved software engineering practices that incorporate integrated analysis and design iterations throughout a model-based development process. The research investigated both the static and dynamic characteristics of the selected techniques and tools, identified characteristics of ACG tools with potential impact on safety, and considered the semantic consistency between representations.

1 Introduction

Software development employing Model-Based Development (MBD) and automated code generation (ACG) from design models has the potential to enhance both the quality of the resulting system and efficiency of the development process. However, safety- and mission-critical systems in aviation, aerospace, medical equipment, transportation, nuclear, and weapons system applications require rigorous verification and validation, often under explicit guidelines and standards defined for the application domain. The suitability of ACG techniques for the implementation of such systems within regulated industries is still being discussed. This paper presents the results of investigations into the engineering challenges associated with the application of ACG technologies, specifically focusing on issues related to ensuring the correctness of the code for safety-critical applications.

The main objective of this work has been to investigate and compile software engineering observations on the use of ACG techniques and tools for safety-critical software development using case studies. The compiled data highlight important characteristics of MBD methodologies and the ACG tools that support them. The research explored the characteristics of model-based software development methodologies and the automated code generation tools that support them. Specifically, data related to the engineering challenges, skills, and effort associated with ACG practices and technologies were collected. These data can be used to determine whether ACG is a viable approach for the creation of safety-critical systems and whether ACG can provide improved efficiency and effectiveness in design, verification, and validation. The case studies involved software engineering practices that incorporate integrated analysis and design iterations throughout the MBD process.

The research focused on the implementation of generated code on a real-time target processing environment and assessing static and dynamic properties of the generated code. Static analysis involves inspections and reviews of the generated code files. Dynamic analysis uses tools to ascertain the executable run-time parameters. The collected metrics included the number of files generated, lines of code generated, number of functions generated, readability of code, footprint, number of threads, execution time, etc. These data can be used to

determine whether ACG is a viable approach for the creation of safety-critical systems and whether ACG can provide improved efficiency and effectiveness in design, verification, and validation.

2 Background

There is a growing tendency in software systems development to work at higher levels of abstraction and to integrate design and analysis, employing models (analyzable formulations) as principal design representations and sophisticated development environments (tools). These model-based approaches rely on building conceptual models of a software system and analyzing those models (via simulation, animation, and semi-formal or formal model checking) before translation into a conventional programming language format. Consistency of the requirements and correctness of specifications are critical issues explored in literature [4,5]. Model-Based Development languages have became the de facto *Lingua Franca* of modern software engineering. Modern software design tools typically address the use of modeling throughout the development lifecycle. These practices rely on the capabilities of Integrated Development Environments (IDE) to produce target software (e.g. automated code generators and compilers).

Software IDE based on various abstract modeling languages (e.g. UML, state machines, finite automata, CSP) are widely used for software development in diverse applications [2]. In addition, they offer the potential to reduce or eliminate the software construction phase of development by providing automated code generation capabilities. However, the use of these technologies for the development of safety-critical systems presents formidable engineering challenges and design concerns. The issues of assurance and correctness of code generation, specifically as applied to safety critical systems have been explored widely in literature [1,3,7,11,13,14].

The introduction of ACG technologies and practices into safety-critical applications must provide support for traceability throughout the development process and design transparency (visibility and clear definition of the formal foundations for design representations and the transitions among them) at a level sufficient to enable effective independent review and certification. Automated systems often limit the visibility into the engineering processes, specifically restricting observation by human reviewers to pre- and post-transformed artifacts. Sometimes visibility is augmented with narrow access to interim representations. However, the rationale and supporting formalism for transitions are often not evident to a reviewer. For example, in tracing/reviewing the transition of a design from source to the machine code produced by a compiler, only the input source text and output compiled source text are available to a reviewer. Occasionally, (e.g. multi-pass compilers) there is some visibility into intermediate representations. However, the core compiler algorithms and optimization strategies are not evident in the design artifacts and must be determined from tool user guides, on-line help, etc. A misinterpretation of compiler transformation algorithms may result in a failure to recognize design errors.

In the case of transforming models to source code traceability and transparency are equally and perhaps more challenging. In this context, the challenges involve concerns centered on semantic precision as well as explicit design transparency, and traceability from requirements through to executable runtime modules. Between each representation: modeling language, source code, and executables, in addition to a syntax discontinuity, there is a semantic boundary where abstractions in one phase must be translated into constructs in another. To ensure correctness, the transformations across these boundaries, as manifested syntactically and semantically, must be explicit and unambiguous. For effective design, this is a bi-directional concern (e.g. to ensure the consistency changes made in source code must be reflected in the modeling language). A precise and explicit definition of the interfaces at the boundaries will ensure traceability and visibility in verification and validation for safety-critical application systems.

As models are translated into source code, particularly for real-time safety-critical systems, a significant challenge is to ensure unambiguous, traceable, and transparent mapping of a modeling language's constructs and constraints into those of source code. In addition to this application model mapping, there must be an unambiguous, traceable, and transparent mapping to a target runtime environment. This execution model mapping involves both general and target specific considerations.

3 Automated Code Generation

In Automated Code Generation well-formed input representations (models) are transformed into "source text" using a tool with ACG functionality to facilitate the transformation. The process is conceptually comparable to one used in compilation of the source code to machine representation [12]. For the ACG, a well-formed representation may be a set of UML class diagrams, state transition diagrams, an architecture description language model, or a variety of other modeling artifacts. Various levels of support for this transformation can be provided. The target source code languages (e.g. Ada, C/C++, Java) can be compiled into machine code for download and execution within an application environment.

ACG can be considered as the next logical step in specifying and describing a software system. In earlier computing eras, developers used binary machine language to represent the required software functionality. Subsequently, assembly language provided the developers with more human-readable form (which was automatically translated into binary code). Another level was introduced with high-level languages (automatically translated into binary through compilers/interpreters). Currently, the use of formalized modeling languages in form of diagrams and other visual representations provide higher levels of abstraction in software engineering. This enables developers to focus on the vital elements of the system rather than on mundane coding. Code generators handle the translation from graphical model to high-level language representation. The level of abstraction is now raised to the point where the system/software architects and designers can contribute more to the engineering solution.

MBD tools support two basic development approaches: structural and functional. The former is more often associated with a software engineering viewpoint where a system is defined in terms of its structure and behavior. The most popular paradigm used is that associated with UML notation. The later is related to control engineering viewpoint wherein a system is defined as sequence of blocks, each realizing one of the system functions. In each of the representations the behavior is defined as sequence of state transitions. *Matlab/Simulink/RTW, RoseRT, Rhapsody, Tau, Artisan Studio, Esterel Studio, RT Builder. Sildex, STOOD, Beacon,* and *SCADE* are examples of such tools. A majority of these tools include some ACG functionality.

Code generators may be text-based or graphical. Some of the tools are capable of fully automated code generation directly from model artifacts. Some use a model to produce a code framework and require the developer to manually enter much of the detailed implementation code. Several tools are based on a proprietary internal language, a formal notation supporting a specific code generation paradigm.

The tools allow a user to create models of the software structure and behavior that the tools convert into modules of source code. The accepted approach is to create dedicated modules for each graphical block that could be used in the model. The generated program instantiates the macros and stitches together the inputs to the outputs of the blocks. The code generation needs to be analyzed to show direct mapping of the model as entered by the designer and the generated code. For many modern tools one needs to select only a subset of the available modeling components to assure easy mapping to the generated code. It is necessary to analyze the limits and constraints in terms of the number of components, connections, hierarchy levels, etc.

Despite widespread use of ACG tools, such issues as inconsistency of the behavior of the modeled system due to the nature of the model representation [10] and effectiveness of the ACG tools use in safety critical systems [8,9] still require further investigation.

4 Project Methodology

A Power Boat Autopilot (PBA), a system that controls the dynamics of a small powered watercraft, was selected as the case study for the project. The PBA involved concurrency, real-time constraints, and closed-loop behavior as well as safety-critical requirements. To implement the system, three commercial tools with ACG functionality were selected. The selection was based on the tools availability, level of use in industry, and modeling approach. The tools represented both object-oriented (UML 2.0) and structured approaches but all three are capable of producing C-language code.

Common software architecture for the PBA was created (Figure 1). Three developers, graduate students of software engineering program, used *Statemate 3.3, Rhapsody 5.2* and *Matlab 7.0.1* to implement this common design and generate C-language code. Each developer was responsible for modeling the system, generating,

compiling, and downloading the code to a real time target environment, and conducting the planned static and dynamic analysis on the model.

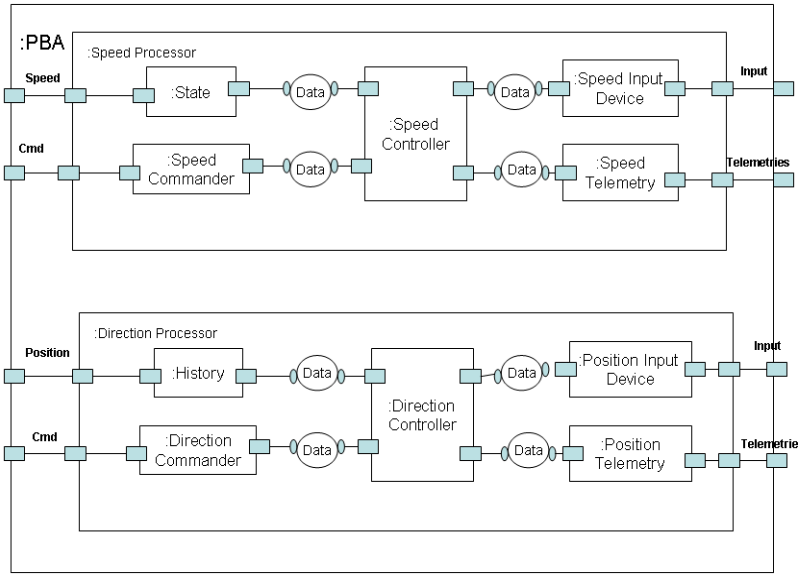

Fig. 1. Case Study Common Architecture

The primary representation of the common architecture was different in each of the tools. In *Matlab*, Stateflow diagrams and finite state machine representations were used to model the system. In *Statemate*, the common architecture was mapped into an activity diagram. In *Rhapsody for C*, class diagrams are required to represent a top-level view of the system, even though the tool does not generate object-oriented code. While there are differences in their modeling approaches, all of the tools generate source code in the C language.

A combination of static analysis and dynamic exploration of the run-time execution were used in an assessment of the automated code generators and their interoperability with integrated development environments. Static inspections and reviews of the code generated by the tools were conducted. These provided quantitative and qualitative (including readability, organization, modularity) information about the generated code. The quantitative metrics (e.g. lines of code, number of functions) were determined from the code. The code inspection also provided information on real-time and concurrency constructs. For instance, scheduling approaches (priority based or cyclic execution) can be identified in the code. Multithreading and concurrency, synchronization and mutual exclusion could be explored and identified in the code.

Dynamic analysis included exploration of the run-time environment. The generated code was compiled and downloaded to a real-time target and executed. Some characteristics of the executable (footprint, number of threads, timing) were assessed by tool-supported observation of the run-time behavior, monitoring the presence of multitasking, and assessment of priorities, memory utilization, etc.

5 Considerations in Safety-Critical Systems

The basic premise of this phase of research has been to identify the features of the tool with an impact on safety of the target application. Table 1 captures the tool features related to the safety of the generated software product. Each feature is identified with appropriate questions to be asked of a tool vendor when selecting a specific tool for use in safety critical application.

Table 1 . Tool Features Supporting Safety

Category	Description/Question
Exception Handler Design Support	Does the tool facilitate creation of exception handlers? Is the implementation of those handlers visible to the developer?
Fault Tolerance Constructs	Does the tool support or facilitate creation of fault tolerance constructs?
Safety Components Library	Does the tool have any pre-built component libraries with components applicable to safety critical applications?
Range Enforcing	Does the tool support, in design and/or runtime, range limitations on variables?
Message Passing	What message passing and communication methods does the tool use? Are they visible to the developer?
Formal Language	Does the tool use a formal language? If so, what it is it? Is it a standard?
Legacy Import	Can the tool import legacy code or integrate with it in some way?
Real time Support	Does the tool have any timing constructs? Does it facilitate testing the meeting of deadlines?
Analysis Support	What support does the tool include for analysis of the models?
Syntax Checking	What syntax checking capabilities does the tool have?
Loop Condition Checking	Does the tool check for infinite loops?
Deadlock and Livelock	Can the tool detect deadlock or livelock in the behavioral models?
Dead Code Checking	Does the tool check for dead code or unreachable states in state machines?
Deterministic Checker	Does the tool check the behavioral models for determinism?
Memory Usage Prediction and Analysis	Can the tool predict overall memory usage? Can it track actual memory usage?
Memory Protection	Does the tool allow for memory protection and separation of processes in the design?
Performance Prediction and Analysis?	Can the tool predict performance? Can it track actual performance?
Simulation Based Analysis	Does the tool have simulation capabilities?
Extended Simulation Support	Does the tool support long-term and multi-run execution simulation?
Simulation Coverage Tracker	Does the tool track coverage in the simulation, inputs, outputs, conditions, paths, etc.?
Stress Testing Support	Does the tool perform stress testing?
Fault Injection Analysis	Does the tool support fault injection?
Testing Support	Does the tool integrate with any testing tools?
Hazard Analysis Tools	Does the tool integrate with any hazard analysis tools?
Requirements	Does the tool integrate with any requirements management

Category	Description/Question
Management Tools	tools?
Reliability Analysis Tools	Does the tool integrate with any reliability analysis tools?
Traceability	Does the tool provide any traceability between requirements and models?
Requirements Validation	Does the tool support verification or validation to check models against requirements?
Compiler Selection	Can the developer to choose a specific compiler to work with the tool?
Compiler Integration	Is the tool flexible to allow developers select the compile options?
Code Generation Options	What options does the tool give on the structure or method of generating the code?

6 Experimental Results

The use of MBD allowed system structure and functions to be captured quickly and enabled the production of easily understood models. The models facilitated communication of key architectural issues among team members. Team members could help each other since the representations of the system were straightforward, even though team members used different tools and methodologies. Problems with the system were quickly found and eliminated using the analysis features of the tools.

Each tool required varying levels of detail, yet all demanded too much detail for the architectural level. For example, when creating a class with one tool, the developer is required to express the exact type of an attribute (even though it may not be known or prudent to decide upon the type so early in the design). The separation of concerns and judicious postponement of design decisions are not strictly enforced in the tools.

A common thread among all of the tools was that a visual representation of a system, at any given point, may or may not represent the complete model of the system resident in the tool. However, at times a developer may be convinced that what is being shown is everything that exists. This may lead to misunderstandings and errors in the system design. An example is creating an association between classes. When the association line is drawn, changes occur to the model and the classes are associated. If that line is then deleted in the visual representation, the only change is that the line is no longer shown in the visual presentation. The association and all that it implies still exist in the system model unless it is removed via another explicit action. This demonstrates the need for a sound knowledge of the tool.

For qualitative measures code readability, traceability to the model, location and content of comments were used. For quantitative analysis the number of lines of code, total size of object file, and the number of functions generated were measured. For this particular example, *Statemate* generated the least amount: 680 LOC, with *Rhapsody* and *Matlab* generating 1,356 LOC and 3,656 LOC respectively. In addition, many of the comments in the generated code did not clearly explain the original design constructs. For example, they specify the beginning and end of files. To fully understand the generated code, a user must understand how each function is generated, since this information is not provided in comments. Table 2 summarizes these results.

Downloading executable code to a target system for each of the tools involves different procedures and requires substantial knowledge of the target execution environment (e.g. the target operating system). Two of the tools allowed the user to connect and run directly from them. Another required running the code directly from the target using an addition to the dedicated target. One key lesson learned in the research is the importance of interoperability. Each of the tools should have clearly defined and working interfaces with other tools and environments (compilers, linkers, loaders, operating system etc.).

All the tools provided some level of transparency to a runtime environment. For example in one case, by analyzing the model the number of tasks running on the system could be identified. Similarly, in another case, when running the software on a real-time target and using a logic analyzer it was clear what concurrent threads corresponded to the active objects defined in the model. In others, the user cannot see what task corresponds to what active class. In general low level transparency is much less effective. For example, when trying to

determine when individual functions are called, started or stopped, the user cannot see these events. In general, a supporting tool was required to identify the code execution sequence.

Table 2. Aggregate Results of the Analysis

	Statemate 3.3	Rhapsody 5.2	Matlab 7.0.1
# of .c and .h files	**4 (.h) and 3(.c)**	**5 (.h) and 5(.c)**	**6 (.h) and 3(.c)**
Readability			
Consistent naming	**Yes**	**Yes**	**Yes**
# of Comments	**Low**	**Low**	**Low**
Comment rationale	**Yes**	**Yes**	**Yes**
Location of Comments	**End of functions and files**	**Per attribute and methods section**	**Start of every function**
Explanatory	**No**	**No**	**No**
Model to Code Constructs	**Evident**	**Evident**	**Evident**
LOC	**680**	**1356**	**3656**
Concurrency Constructs	**No**	**Yes**	**Yes**
# of Tasks Spawned	**1***	**3**	**3**

For *Matlab*, the number of tasks in the target software is equal to the number of different sample time rates defined for Stateflow blocks expressed as a statechart in the model. It may have a separate thread of execution associated with each thread. However, a block cannot have multiple sub threads for parallel states inside the statechart. *Matlab* requires specific parameters to be set during model configuration to generate code handling multitasking. The tasking mode in configuration settings needs to be set to multitasking. An interesting observation while experimenting with this facility was that if the model does not have different sample rates, the multitasking functionality does not work. It gives a compilation error requesting to use at least two different sample rates.

One general observation is the importance of documentation. The documentation must be very specific and well understood, especially regarding how the code generator works and how to interface with other environments and tools. This is the key for efficiently using the tool and in understanding, manipulating, and analyzing the generated code.

The developers recorded their efforts using Personal Software Process (PSP) techniques [6]. Effort was categorized as: modeling, compiling/downloading, or static/dynamic analysis. The percentage assignment of this effort to the three development areas is presented in Table 3.

Table 3. Effort Analysis

TOOL	Modeling	Compiling and Downloading	Static and Dynamic Analysis	Administrative
Statemate	15%	39%	16%	30%
Rhapsody	10%	42%	15%	33%
Matlab	20%	50%	12%	18%

It is interesting to note the relatively high effort on creating the code and downloading to the target. While tools make the model development and analysis activities quite user-friendly, the creation of executable code requires good knowledge of run-time environment and the idiosyncrasies of the host-target. We anticipate that for more experienced users of the tools, the percentage of time in the compiling and downloading category

would be less. Many of the problems encountered in this case study resulted from mistakes with and misunderstandings of the tool. This situation was not unexpected, since the researchers did not have experience with the tools prior to the start of the project.

A number of technical problems occurred during the compiling process with the tools. In one case, it was possible to generate ANSI C code from the model but the generated make files did not work. This resulted mismatches of syntax, for example '/' is used as path recognition instead of '\' and options are identified by '\opt' and not as '-opt' (therefore options are treated as folders and not as compiler options). Once these problems were identified and addressed, scheduler libraries were recognized.

7 Conclusions

Increasingly, handcrafting of source code is being replaced by more sophisticated development practices (e.g. MBD) that are supported by tools with some form of automated code generation capabilities. Features of these practices and tools that ensure determinism, correctness, robustness, and conformance to standards are crucial to overall system quality. In addition for the development of safety-critical systems, traceability and transparency of the design and implementation artifacts throughout the complete engineering process are vital to complying with industry and certification standards. In adopting new technologies or practices, management and technical members of organization must recognize these requirements.

Model-based development techniques and the automated tools (in particular ACG tools) that support them must provide the means to facilitate and document traceability and provide undistorted visibility into the design and implementation artifacts. There is a semantic mapping at boundaries of the progressive representations of the system, from the model to implementation. This mapping must be unambiguous (deterministic) and there must be visibility (transparency) into this mapping. A tool's capabilities in ensuring traceability between artifacts generated from one development phase to another can be a starting point to make the arguments about tool quality and usability in a regulated industry.

Different skills and knowledge will be required for adoption of the MBD practices. The expert at abstract modeling (abstractionists) will be a dominant professional within MBD practices, supplanting central position held by a skilled programmer in manual code development practices. These abstractionists understand the foundations of software systems architecture, modeling techniques, the implication of the selected solution on the resulting code, and the relation between the abstract development environment and the runtime execution environment of a target system.

Acknowledgement

The authors acknowledge the support of the CRM Guidant and the contribution of Stefano Grimaldi, Soma Mitra, and Sona Johri, graduate students of ERAU software engineering program, in the research leading up to this paper.

References

1. Denney, E., Fischer B., Schumann J.: *Adding Assurance to Automatically Generated Code,* Proceedings of Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE'04), March 2004
2. Federal Aviation Administration, Software Tools Forum, Embry Riddle University, Daytona Beach, Fl., May 2004, `http://www.erau.edu/db/campus/softwaretoolsforum.html`
3. Halbwachs N., Raymond P., and Ratel C.: *Generating efficient code from data-flow programs.* Third International Symposium on Programming Language Implementation and Logic Programming , Passau (Germany), August 1991
4. Heitmeyer C.L, Jeffords R.D., Labaw B.G.: *Automated Consistency Checking of Requirements Specifications.* ACM Transactions of Software Engineering and Methodology , **5** (3):231–261, July 1996.
5. Hohman W.: *Supporting Model-Based Development with Unambiguous Specifications,* Formal Verification and Correct-By-Construction Embedded Software, SAE World Congress, Detroit, MI, March 8-11, 2004
6. Humphrey W.: *Introduction to the Personal Software Process,* Addison-Wesley, Reading, Mass.1994

7. Keenan D.J., Heimdahl, M.: *Code Generation from Hierarchical State Machines.* Proceedings of the International Symposium on Requirements Engineering , 1997.
8. Kornecki, A., Zalewski, J.: *Experimental Evaluation of Software Development Tools for Safety Critical Real-Time Systems,* NASA Journal Innovations in Systems and Software Engineering, July, 2005
9. Kornecki A., Zalewski, J.: *Assessment of Software Development Tools for Safety Critical Real Time Systems.* Invited Paper in IFAC Workshop on Programmable Devices and Systems, Ostrava, Czech Republic, February 2003, pp. 2-7
10. Kornecki, A., Erwin, J.: *Characteristics of Safety Critical Software,* Proceedings of the 22nd International System Safety Conference, System Safety Society, ISBN 0-9721385-4-4, Providence, RI, August 2004,
11. O'Halloran C.: *Issues for the Automatic Generation of Safety Critical Software,* 15th IEEE International Conference on Automated Software Engineering (ASE'00), 2000
12. Stepney, S.: *High Integrity Compilation.* Prentice Hall, 1993
13. Vestal, S.: *Assuring the Correctness of Automatically Generated Software.* AIAA/IEEE Digital Avionics Systems Conference , volume **13** , pages 111–118, 1994.
14. Whalen, M.W., Heimdahl, M.: *An Approach to Automatic Code Generation for Safety-Critical Systems,* Proceedings of the 14th IEEE International Conference on Automated Software Engineering, Orlando, October, 1999.