A Study of Automatic Code Generation for Safety-Critical Software: Preliminary Report

Lazar Crawford, Jared Erwin, Stefano Grimaldi, Soma Mitra, Andrew J. Kornecki, David P. Gluch
Embry Riddle Aeronautical University+
Daytona Beach, FL
<gluchd@erau.edu>

Introduction

Modern safety-critical systems (e.g., combined pacemaker/deliberator devices, distributed patient therapy delivery systems) incorporate more functionality than similar devices of the past. The development of these complex systems challenges existing quality assurance techniques; results in significantly longer development times; and demands greater staffing resources to ensure quality and timely product completion.

This is an interim report on a case study of the efficacy and viability of Automatic Code Generation (ACG) techniques applied in the development of real-time, safety-critical software-dependent systems [1]. The research uses Model-Based Software Engineering (MBSE) practices that incorporate integrated analysis and design iterations throughout the development process. The focus of these investigations is the application of automated code generation tools that embody various methodologies, in the development of safety critical systems. There was no attempt to embark on explicit tool comparisons or evaluations.

Automatic Code Generation

Automatic Code Generation, simply described, is a set of well-formed input representations (models) transformed into "source text." An ACG tool facilitates the transformation. A well-formed representation may be a set of UML class diagrams, model-based statecharts, an architecture description language model, or a variety of other modeling artifacts. Target languages also come in a variety of forms, including high-level computer languages (e.g. Ada, C++, Java) [2].

Tool Selection

Tool selection criteria were established and used to identify appropriate tools for the study. The selection criteria address general development capabilities, specific real-time, safety-critical problem domain considerations, and the methodology underlying the tool. Some of the criteria used to evaluate the tools considered issues such as: languages supported, design methodology, capability for complete or partial code generation, real time design capability, and analysis (i.e. simulation, static checking, etc.) and testing capabilities. In this context, several tools were reviewed using a tool-criteria matrix including *Scade*, *Statemate*, *Tau 2.2*, *Rhapsody*, *Stood*, *MatLab/Simulink*, and *Rational Rose RT*.

The tools for this initial "review" were chosen based on availability, widespread use on industrial project, and evidence of real-time development capabilities. The matrix was examined and, through team consensus, three tools were chosen from the initial set. The tools selected were *Statemate, Rhapsody, and Tau 2.2*.

Research Methodology

The project is divided into two phases. Each phase involves instrumented (measured) investigations of the application of the selected techniques and tools. Phase 1 focused on learning. Throughout phase 1, timing data, similar to Personal Software Process (PSP) information, was recorded and engineering observations were made regarding tool use, modeling capabilities, and safety characteristics [3]. These data form the basis for analyses of the effectiveness and viability of the various tools and provide a foundation for defining phase 2 efforts.

As a technique to reduce the impact of the learning time, the initial phase involved (1) the completion of basic tutorials associated with each tool and (2) the development of a simple problem—a car alarm system. The tutorials provided a basic knowledge of the tools and the car alarm system enabled investigators to become more proficient in their use.

The car alarm problem was chosen because it is a reactive system with timing considerations. It is

+Support for this work is provided by the Cardiac Rhythm Management Division of the Guidant Corporation and the Guidant Foundation.

complex enough to learn the tool, yet simple enough to complete relatively quickly. Starting from a common set of requirements, architecture, and context diagram for the system, the car alarm was implemented. Researchers could, therefore, evaluate needed steps from "Concept to Code" and achieve a better understanding of some of the idiosyncrasies of automatic code generation.

Phase I Preliminary Tool Observations

Several preliminary observations highlight how an ACG tool can expedite the development process.

The tools impose consistency and help to avoid basic syntax and referential errors. For example, *Rhapsody*, which uses UML 1.4, automatically checks all added methods, calls and references. It adds accessor methods for attributes added to a class. It also dynamically creates the code as the developer creates diagrams. This allows the developer to switch back and forth, constantly maintaining consistency between models and code; changes in one automatically reflected in the other.

Analysis capabilities of the tools can detect errors in design and implementation early. For example, *Statemate* uses statecharts to define system behavior. The statecharts can be easily created and simulated without any knowledge of code. These statecharts can then be visually simulated. In addition to the graphical representation, a history of all system behavior is recorded for further analysis. A model checker available with *Statemate* includes the ability to check the statecharts for deadlock, non-determinism, and more. Data management is also easily examined and controlled via the Data dictionary in *Statemate*, providing insight into the systems data and a means for handling scope.

Differing, yet connected, models allow developers to examine the same system from varying points of view. In the case of *Tau 2.2*, it supports all the models specified by UML 2.0. This allows the creation of class diagrams to examine system structure, sequence diagrams for runtime analysis, architecture diagrams to view the system communication, and deployment diagrams to analysis runtime entities. All of these diagrams work together, and any change in one is reflected in the others. A simple syntax checker is always running, checking the models as they are being created, so any inconsistencies introduced are immediately flagged.

Summary

The observations taken from phase 1 have been used to formulate the plans for and guide phase 2 efforts. The data from phase 2 will progress beyond a tool focus and provide additional details on the utility of ACG techniques, particularly in a safety-critical environment. To support phase 2 investigations, a set of issues to consider is being developed. These identify specific capabilities and features of the tools, underlying methodology, and associated practices that are important for safety-critical development. Examples include: any support the tool has for creating fault tolerant constructs (watchdogs or n-versioning), facilitating analyses of models for deadlock, or interfacing with hazard/safety analysis tools.

The objective of these investigations is to compile data that will highlight important characteristics of ACG methodologies and tools and to identify the skills, time, and challenges associated with their use. Through these efforts, insight can be gained as to whether ACG is a viable resource with respect to real time and safety critical environments. It is expected that the results will help organizations in their assessment of ACG technology and, as appropriate, help facilitate the transition of these techniques into safety-critical software development practices.

References

[1] M.W. Whalen. "Provably Correct Code Generation for Safety-Critical Systems" *Proceedings of the IEEE International Symposium on Requirements Engineering*, Annapolis Maryland, January, 1997.

[2] M. W. Whalen, Mats P.E. Heimdahl. "An Approach to Automatic Code Generation for Safety-Critical Systems" *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, Orlando, October, 1999.

[3] W. Humphrey. "Introduction to the Personal Software Process." Addison Wesley 1997