

Comparison of Object Oriented Technology Automatic Codes Generating Tools for Safety Critical Real-Time Software

Farahzad Behi

**Embry Riddle Aeronautical University
Computer & Software Engineering
Daytona Beach, FL 32114
Phone: 1-386-226-6888
Fax: 1-386-226-6678
behif@erau.edu**

Daniel Penny III

**Embry Riddle Aeronautical University
Computer & Software Engineering
Daytona Beach, FL 32114
Phone: 1-321-446-4383
Daniel.Penny.III@gmail.com**

ABSTRACT

The purpose of this paper is to compare two different automatic code generating software tools and the suitability of each for the development of safety-critical real-time software systems. The tools used are Rhapsody and SCADE for Avionics. Rhapsody will generate software under ANSI C++ standards and guidelines, while SCADE for Avionics will produce DO-178B certifiable ANSI C++ software. A simple system was designed by both tools and the result of the code execution was discussed focusing on the deterministic characteristic of safety critical Real-Time systems.

Keywords

Software Engineering, Software Tools, DO-178B, Automatic Code Generation. Safety critical, Real-TimeSoftware Process.

1. INTRODUCTION

As the use of computers is dominating all industries from medicine to defense, developing real-time software that is safe, reliable, and predictable is critical to surviving in the future, especially in the aviation/aerospace competitive arena. Safety-critical real-time software systems have different characteristics from general-purpose software such as word processors, databases, etc. Safety-critical real-time software must produce correct results (functional requirements) within specific time restrictions or envelopes (response time requirements). It must be deterministic, and most importantly, it must not fail.

“Safety-critical software is any software that can directly or indirectly contribute to the occurrence of a hazardous system state” [2]. These systems must be safe, reliable, and predictable. The quality of service of these systems depends on how safe they are, how well they can cooperate with hardware, how testable they are, and how dependable they are. Using different software development methodologies and languages in developing these systems could affect many aspects of the systems during runtime.

Today, object-oriented technology is seen by many in the mainstream software community as the “silver bullet” that will take us into the new millennium of software development [1]. Despite this realignment, safety-critical application designers tend to favor a proven technology over having to verify a new approach. Since OOT has proven to be cost-effective and technically sound for many projects of a non-safety critical nature, manufacturers of safety-critical systems are now considering its use [1] with some reservations. The problem is attempting to use object-oriented technology without any

consideration to dependency, reliability, safety, and most important determinism during runtime and risking a catastrophic failure.

To resolve this problem and minimize the risk one could consider the simplest means: follow strict standards and guidelines that prevent the type of problems that would affect a safety critical application. A non-safety critical system developer may use multiple instances of a single object to utilize the full breadth of the object oriented language, but this would not be deterministic during runtime. During execution, there is no way to determine which instance of the object is which. Careful analysis of the assembly language may reveal which instance is being used by its address, but this could be difficult and an inefficient waste of time during verification and certification stages.

The Radio Technical Commission for Aeronautics (RTCA) has published an internationally accepted standard addressing object-oriented technology in civil aviation projects. This standard is known as DO-178B, "Software Considerations in Airborne systems and Equipment Certification." It is the primary application for manufacturers to approve software within airborne systems.

This paper will discuss the result of an experiment comparing two different automatic code generating software tools for safety critical real time software and compares DO-178B certifiable code verses standard code. The tools used are Rhapsody and SCADE for Avionics. Rhapsody will generate software under ANSI C++ standards and guidelines, while SCADE for Avionics will produce DO-178B certifiable ANSI C++ software.

2. DO-178B BRIEF OVERVIEW

For the benefit of the readers who might not be familiar with DO-178B standards, here is a brief overview of it. This standard provides any programmer, company, or visionary with the acceptable limits that aviation software is required to adhere to in order to achieve flight worthy status. All software, regardless of its intention or size has an impact on its intended flight vehicle (rocket, shuttle, airplane, UAV) to some degree or another. This impact is known as a certification level in DO-178B with respect to the aircraft, its crew, and its occupants.

There are five software certification levels under DO-178B procedures. These levels are A through E, where A is the highest impact and E is the least impact. Level A, also known as catastrophic failure, represents software that would "prevent continued safe flight and landing of the aircraft," such as a failure in the landing gear control systems software. Level B, also known as hazardous failure, represents software that if failed would reduce the capabilities of the aircraft forcing the crew to adjust to the reduction in functionality. As an affect of a Level B failure, the aircraft may operate under reduced safety margins, an increase in the crew workload, and/or potential impact on some of the aircraft occupants. A Level C software failure will also reduce the capability of the aircraft and increase the workload of the crew to account for the loss as in Level B. The difference between B and C is that the impact is less than that of a Level B failure to aircraft occupants and crew. A Level D failure represents a minor condition that would not affect flight capabilities or impact the crew's actions but would render some functions impossible, such as an internal health monitoring capabilities ceased to function. Level E software is non-critical to any particular aspect of maintaining flight characteristics or degrades the overall capabilities of the software, such as internal data logging of health status for potential diagnosis purposes.

Some of the issues programmers must address for the development and implementation of DO-178B certified OOT in airborne software are: traceability, coverage, dead code, deactivated code, inheritance, dynamic memory allocation, and most importantly determinism.

These issues affect a safety critical system under development using DO-178B certification rules a great deal. Careful consideration of each issue with respect to DO-178B will produce a safety critical design that will be not only being safe, but highly deterministic. Determinism is the key to a successful software build and a safe future for those using the end product.

3. EXPERIMENT

The following experiment focused on determinism in C++ and how automatic code generation tools of today are taking into account the rigors of safety and reliability. Using a simple model problem of a vehicle cruise control, the development of models for Rhapsody and SCADE shall demonstrate the effectiveness of auto code generation on OOT. Using ANSI C/C++,

Rhapsody shall generate code to simulate the ANSI C/C++ world without DO-178B certification in consideration. SCADE shall produce a similar model of the same system but with ANSI C/C++ DO-178B certified.

Both auto-generated code sets will be compared and contrasted on many levels and aspects, but most important is runtime. Time stamps at the beginning and end of each cycle will determine how effective DO-178B is or isn't in comparison to Rhapsody alone. Execution times will provide the necessary comparison that will allow the side by side demonstration on an Texas Instrument DSP (Digital Signal Processor) target running Code Composer as the operating system.

4. METHOD

Using SCADE and Rhapsody, generate models that will implement the same system. This system will be a real-time safety critical application that executes on a periodic basis or has a periodic cycle. Using a non-impacting method, measure the execution times of the periodic function and compare the average execution times between the generated code by SCADE and Rhapsody respectfully.

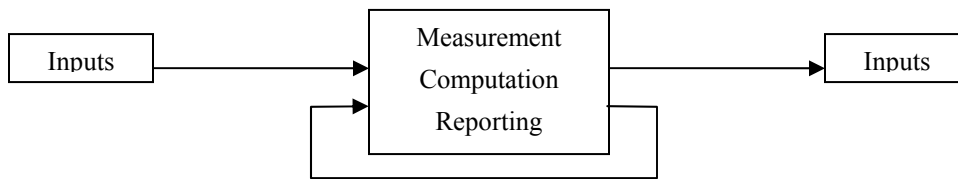


Fig. R 1 – System Conceptual Diagram

4.1 Non-Impacting Measurement Method

Two time tags will determine the system time that the system's periodic behavior executes. Using a results array located within the class that initiates the periodic behavior, the class will store the difference between a starting and ending time tag. The location of these time tags is critical in reducing error in the measurement. The main function will store the starting time immediately before initiating the periodic behavior. The same function will store the ending time tag immediately when the periodic cycle is complete. The difference is immediately calculated and stored in the local results array. Having the time tags immediately before and after the code under investigation allows for the error in the time tag easily removed and management properly.

The time necessary to store the current time, such as in the starting time tag, is determined by the summation of the execution times of each instruction required to perform the store operation. Since this error will remain constant for any combination of experiments, the resulting execution time will include the time needed to get the current time at the end of each cycle. The start time does not impact the execution time unlike the end time.

5. REAL-TIME SAFETY CRITICAL APPLICATION

The system under analysis is a safety critical system found in almost every automobile today. The cruise control is considered fundamental in the realm of real-time safety critical applications but it demonstrates the deepest and basic principles of a real-time safety critical system [3]. The cruise control example below handles changes in the system's environment from the human driver and the vehicle. Once the driver sets the desired speed, the vehicles current speed is compared with the desired and action is taken to conform to guidelines and limitations that defining the cruise control system.

The following model representations are very different in how the cruise control system is designed and implemented, but investigation of auto-generated code will yield two comparable software systems. The SCADE approach is highly graphical using Boolean algebra control logic instead of state machines. Rhapsody follows strict UML practices and requires input from the user in order to complete the code generation similar to that of SCADE. The following diagram represents a

conceptual view of the cruise control system used for development of both models in SCADE and Rhapsody. This cruise control system is feedback control loop system that acts on various external events and set desired speed input.

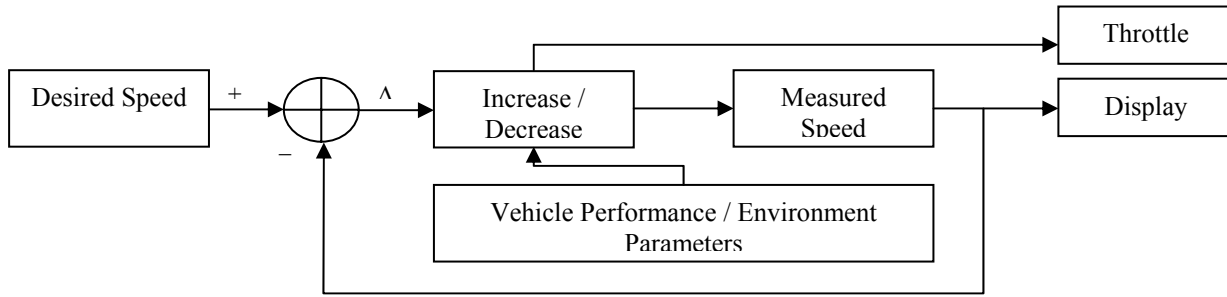


Fig. R 2 - Cruise Control System Feedback Loop [3]

5.1 SCADE Model

The following cruise control system is produced using SCADE [3]. It is based on an example provided by Esterel Technologies that has been modified to produce DO-178B certified development. In order for SCADE to generate DO-178B certified code, all paths must be deterministic. This requires that all state machines be defined using boolean control logic.

The DO178B_CruiseState and DO178B_SpeedFilter are designed with DO-178B certification requirements in mind. State machines are not compliant with the DO-178B code generator for SCADE. Therefore below are the boolean control logic statements that implement the state machine describing the cruise control operational states.

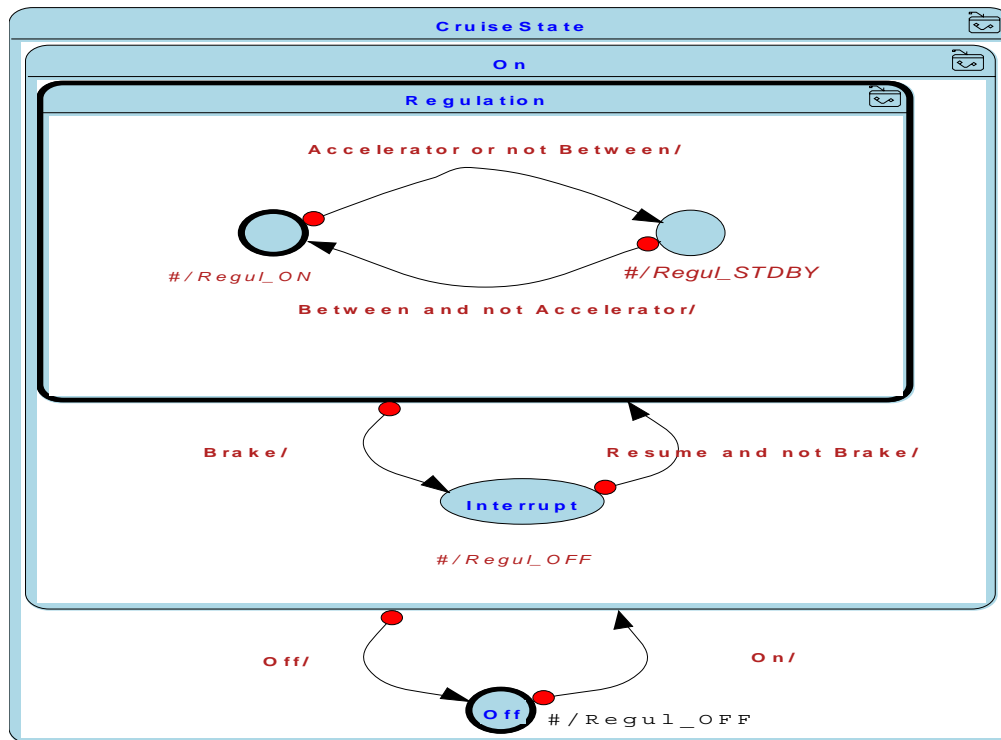


Fig. R 3 - Cruise Control State Machine

$Regul_OFF = Brake \text{ OR } (OFF \text{ and } !ON)$

$Regul_ON = (ON \text{ AND } !OFF) \text{ AND } (Resume \text{ AND } !Brake)$

$Regul_STBY = !(Regul_OFF) \text{ AND } (Accel \text{ AND } !Between) \text{ AND } (!Brake)$

5.2 Rhapsody Model

Rhapsody 5.2 uses strict Unified Modeling Language (UML) and internal working functional details to populate its auto-generated code. The model that accurately represents the cruise control system is represented below in an object model diagram or class diagram. The central cruiseControl class calls upon its helper classes to perform necessary functionality to meet the same operations as the SCADE model. A closer investigation of the Rhapsody model yields the inner working of each class.

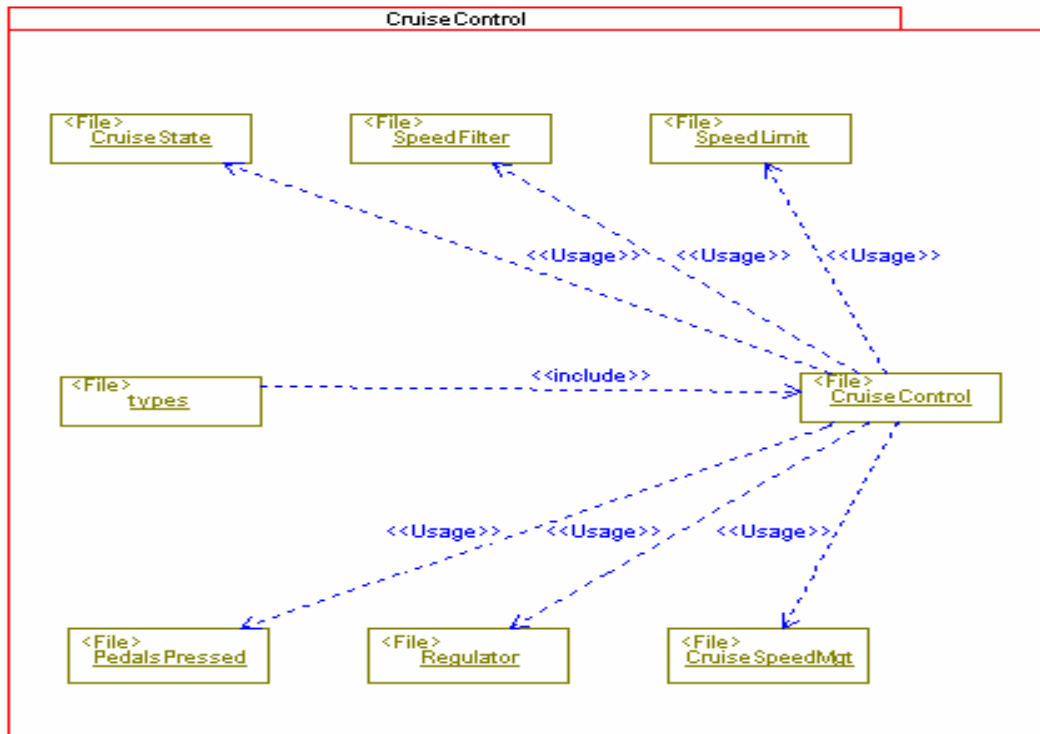


Fig. R6 – Rhapsody Class Diagram

6. EXPERIMENT RESULTS

The results of the SCADE and Rhapsody code yielded an average execution time very similar to each other. Despite their small difference in execution time, the SCADE code performed in a highly deterministic fashion, varying three of one hundred cycles from the average 3.704 milliseconds. The Rhapsody code seemed to be on average shorter execution time of 3.149.

Unexpectedly the Rhapsody model produced the shorter execution time, but this may be due to the additional functionality required to implement the DO-178B certifiable code. The difference in execution times poses a very interesting safety critical concern for the development of small real-time software systems. Should a system require that the software perform within certain limitations, the Rhapsody code may be considered better due to the faster execution and higher response rates. However, should safety be a major concern, the DO-178B certifiable safety critical code should be considered the better choice despite the longer execution time. Knowing that a cruise control for a vehicle is not exactly the same as a commercial airliner, but the principles are the same.

This experiment sought out to show the difference between DO-178B certifiable code versus its standard ANSI counterpart. Interestingly enough, the DO-178B code was outperformed, but not guaranteed to be safe without greater inspection, verification, and validation efforts on the software engineer and independent investigators' response. Is the additional effort worthwhile for the higher execution and certifiable code automatically generated from SCADE. This question will haunt software engineers and architects on every software project in the future. These decisions are based on a project-by-project basis since each software application has different requirements, different environments, and different customer needs.

If the project were to be repeated, additional investigation into a multi-partition operating system that would allow multiple task partitions to operate on a single processor would render a more accurate response on the demands of today's computing world. Most systems operate with two or more applications on a single processor requiring time and space partitioning, resulting in a much more complex. These multi-role systems are a higher focus and demand on the DO-178B certifiable code and would prove highly cost-effective to

implement using a software tool such as SCADE rather than Rhapsody where certification lies within the hands of the engineer and verifier.

Rhapsody Results

1	3.153	21	3.152	41	3.150	61	3.150	81	3.144
2	3.153	22	3.147	42	3.149	62	3.148	82	3.153
3	3.151	23	3.153	43	3.148	63	3.147	83	3.149
4	3.148	24	3.153	44	3.146	64	3.145	84	3.147
5	3.151	25	3.153	45	3.146	65	3.152	85	3.152
6	3.145	26	3.149	46	3.145	66	3.145	86	3.152
7	3.153	27	3.152	47	3.149	67	3.147	87	3.144
8	3.147	28	3.144	48	3.146	68	3.152	88	3.144
9	3.148	29	3.150	49	3.145	69	3.144	89	3.147
10	3.149	30	3.145	50	3.152	70	3.149	90	3.144
11	3.149	31	3.153	51	3.149	71	3.148	91	3.145
12	3.150	32	3.147	52	3.151	72	3.147	92	3.149
13	3.150	33	3.153	53	3.149	73	3.145	93	3.148
14	3.152	34	3.148	54	3.147	74	3.154	94	3.148
15	3.150	35	3.146	55	3.147	75	3.153	95	3.153
16	3.151	36	3.153	56	3.153	76	3.150	96	3.149
17	3.145	37	3.146	57	3.148	77	3.148	97	3.147
18	3.154	38	3.152	58	3.144	78	3.146	98	3.145
19	3.153	39	3.148	59	3.145	79	3.148	99	3.149
20	3.149	40	3.146	60	3.148	80	3.148	100	3.147

Average = 3.149 millisecond

SCADE Results

1	3.704	21	3.704	41	3.704	61	3.704	81	3.704
2	3.704	22	3.704	42	3.704	62	3.704	82	3.704
3	3.704	23	3.704	43	3.704	63	3.704	83	3.704
4	3.704	24	3.704	44	3.704	64	3.704	84	3.704
5	3.704	25	3.704	45	3.704	65	3.704	85	3.704
6	3.704	26	3.704	46	3.704	66	3.704	86	3.704
7	3.704	27	3.704	47	3.704	67	3.704	87	3.704
8	3.704	28	3.704	48	3.704	68	3.704	88	3.704
9	3.704	29	3.704	49	3.704	69	3.706	89	3.704
10	3.704	30	3.704	50	3.704	70	3.704	90	3.704
11	3.704	31	3.704	51	3.704	71	3.704	91	3.704
12	3.704	32	3.704	52	3.704	72	3.704	92	3.705
13	3.704	33	3.704	53	3.704	73	3.704	93	3.704
14	3.704	34	3.704	54	3.704	74	3.704	94	3.704
15	3.704	35	3.704	55	3.704	75	3.704	95	3.704

16	3.704	36	3.704	56	3.704	76	3.704	96	3.704
17	3.704	37	3.704	57	3.704	77	3.704	97	3.704
18	3.704	38	3.704	58	3.704	78	3.704	98	3.704
19	3.704	39	3.704	59	3.704	79	3.704	99	3.704
20	3.704	40	3.705	60	3.704	80	3.704	100	3.704

Average = 3.704 milliseconds

7. EXPERIMENT RESULTS

The world of software is dramatically changing and with vigor. The keystrokes heard round the world will soon be united in a single unison: programming using object-oriented technology in safety critical real-time systems. The development of these systems is only possible with great focus and diligence during each and every stage of software development lifecycle.

This paper proves through experimentation and supporting evidence, the use of internationally accepted object-oriented programming standards, such as DO-178B, improves real-time software development and execution time. This improvement applies to the system development by reducing the total time and the number of resources required to produce object-oriented technology for safety-critical applications due to strict guidelines. These rules governing the use of OOT reduces the amount of verification, validation, and testing required certifying the software for use where human life may be put in harms way.

Using standards such as the one described in this paper and others developed from tried and tested methods assists in reducing the bottom line for any software system while maintaining the highest level of safety. The more intense the market grows for systems that are lighter, faster, cheaper; changes are required to meet the demand on shorter time tables and the only sure way to produce certifiable code is through the use of a OOT approach that requires discipline, focus, and dedication to producing a safety critical application that is deterministic and reliable.

Determinism shall rule the object oriented world as complexity increases and more and more people depend on technology on a day to day basis. The key to a successful safety-critical application is one that can be predicted not only during execution at run-time but also within a timely fashion within the boundaries set by designers and/or the environment the application will operate. The world is at the fingertips of each and every programmer developing safety-critical applications using OOT, object-oriented technology: the new standard.

8. REFERENCES

- [1] Certification Authorities Software Team, position paper CAST-4, "Object Oriented Technology (OOT) In Civil Aviation Projects: Certification Concerns", Completed January, 2000.
- [2] Nancy G. Leveson, Safeware, System Safety and Computers, Addison-Wesley, 1995.
- [3] www.esterel-technologies.com . Esterel Studio user guide, tutorial and reference document
- [4] "Glossary of Software Engineering Terminology." ANSI/IEEE Standard, 1983.
- [5] RTCA, document DO-178B/ED-12B, "Software Considerations in Airborne Systems and Equipment Certification