

Automatic Engine Control Code Generation with Integrated Automatic Static Code Verification

Robert E.L. Harmon, Cummins Engines, Inc.
Chris Hôte, PolySpace Technologies, USA Operations

ABSTRACT

The software design and development of engine controllers evolved into a product line architecture in order to keep up with increasing customers' needs. Additional and reliable functionalities were required to be delivered at lower costs and in a shorter timeframe. The amount of embedded code for Cummins engines has been multiplied ten-fold over the last decade.

Traditional engine controller module development methods cannot handle a still more and more demanding market as those methods involve multiple manual interactions including hand coding, code review, unit/integration tests that are error-prone and time consuming.

Given product line architecture, the viable approach is to use Model-Based Design for creating executable specifications, automatically generating controller code, and performing verification and validation (V&V) activities on the model.

Furthermore, recent static analysis techniques are applied to sub-systems very early in the software design process for finding design flaws leading to software reliability breaches (e.g. arithmetic exceptions). Those techniques are applied before functional tests are performed or later on for final sanity check.

The technology supports links back to the model and applies to different levels of the model, including subsystems and blocks, making it easy to resolve issues, to perform traceability analysis and to support code review effort.

This paper describes a working example of an integrated environment using Model-Based Design with automatic code generation and advanced code analysis tools.

INTRODUCTION

Cummins Inc. is a global power leader that designs, manufactures, distributes and services engines and related technologies, through four complementary business units. Headquartered in Columbus, Indiana, USA, Cummins Inc. serves customers in more than 160 countries, reported \$9.9bns in 2005, and counts more than 33,500 employees worldwide.

Cummins Inc. has an historical and continuous commitment to innovation that has been insuring its market leadership over the years:

- Cummins was one of the first companies to envision the commercial potential of unproven engine technology; Rudolf Diesel invented in the

- early 1900's.
- Cummins was the first company to offer the industry 100,000-mile guarantee.

Cummins Inc. success still relies on innovative ways of thinking that apply deep in engine design and development.

The engine business unit represents about 60% of the company total revenues. It has produced more than 700,000 diesel or natural gas-powered engines that equip a very wide range of vehicles from Dodge RAM pickup truck to buses, marine equipment, and heavy-duty 200 tons mining trucks.

Handling such a wide diversity of systems is a challenging task, which costs reduction and quality improvement levels are monitored through an innovative Six Sigma program. Key techniques including advanced design approach, automatic code generation and code verification constitute the backbone of software development process at Cummins and are explained in the following sections.

BUILDING ENGINE CONTROLLERS AT CUMMINS ENGINES INC.

Several factors may describe the strategic challenges the Cummins engine business faces:

Firstly, Cummins Inc. is a power supplier and needs to answer to market needs expressed through manufacturers of car, buses, trucks, and other systems. In order to stay competitive, Cummins Inc. needs to timely react to customer requirement changes while maintaining minimal production and maintenance costs. For example, an engine truck should successfully operate either in "chilly" Alaskan winter or in Arizona during summer time. This makes a 150F temperature range in which engine performance and reliability is to be insured.

Secondly, Cummins Inc. services engines all over the world in demanding and versatile environments. Engine reliability is then at stake as an engine failure may cost a lot of money and may seriously deteriorate the Cummins corporate image: Just imagine the consequences of a 200 tons truck stuck in an opened-air gold mine in South Africa because the engine cannot start.

Building a reliable and adaptable engine is a complex task, some of them go beyond the scope of this paper (e.g.: mechanical, electrical, environmental challenges). We wish to focus here on the software part of engine controllers.

The main features of the engine controller break down into several categories:

- Reliability: The engine controller must not fail as it mainly leads to availability losses with the financial and corporate image impacts that

derive from it. How then to maintain or to increase quality level of more or more complex software or deal with a limited testing budget?

- Flexibility, versatility: The same engine controller depends on hundreds sensors that must operate in a huge variety of environmental conditions (temperature, humidity, pressure, gas composition). How to validate and verify software that depends on many calibration data that may be changed or fine-tuned while the engine is in operation?
- Reusability: In order to keep development costs low, the engine controller software may run on different target processor and may also be fine-tuned after it has been released. How to measure the reliability of software applications that are either targeted towards 16 or 32 bits processors?

How then to design such highly configurable system in a timely manner while matching high quality level?

AUTOMATED TECHNIQUES TO REDUCE COSTS FOR CODE PRODUCTION AND VERIFICATION

Existing techniques based on manual interaction including code review, white-box testing, unit-integration tests, late detection of errors and cumbersome debugging operations during field tests. These techniques were viewed as inhibitors. The resulting philosophy was to find and resolve errors further down the development cycle. It was felt that all errors would manifest themselves in a component or system test. Although this is true for error detection, the error isolation effort is compounded. Since the opportunity to isolate hundreds of lines of code is now imbedded into hundreds of thousands of lines of code. This was not initially recognized since internal testing was content to remain at the component or system test levels.

So a culture change was required. Just as the product line approach had dictated an abstraction change to the code (maintain Simulink® diagrams and not the C code). The product line approach dictated components and units had to be tested. What good does it do the have dysfunctional interchangeable component architecture? Cummins had to change the testing approach to reacquaint developers with unit test and at the same time not extend the product cycle. This seemingly paradox was solved by using MathWorks and PolySpace products.

Cummins expressed the need to provide the developers a means to unit test and debug the automatically generated code. Since the developers use Simulink and Stateflow® for the design and then automatically generate the code, they have never seen the code they are about to troubleshoot. Being domain experts, they don't really want to debug or test their components at the code level. PolySpace and MathWorks teamed up and enhanced the development tool set. With the new integrated tool chain, the developers can unit test and debug their

components by directly changing the Simulink/Stateflow diagram without being intimately familiar with the automatically generated code.

The picture below indicates how the products collaborate:

- On the left side of the picture is one Simulink block. The PolySpace product is activated through Simulink.
- On the right side, the generated code is displayed along with the PolySpace diagnostics. An orange code section indicates a design breach and is linked to an operator in the Simulink model. The green sections show reliable operations under all circumstances.
- The picture shows an overflow may occur in the code (red square, overflow on "Gain1"). By looking back to the Simulink model, it appears the "k1" gain operator is to be considered. Indeed, the breach is caused by the fact an external data is directly connected to the gain. Depending on external data value the overflow may happen and therefore a protection should be added here (e.g.; Limiter operator).

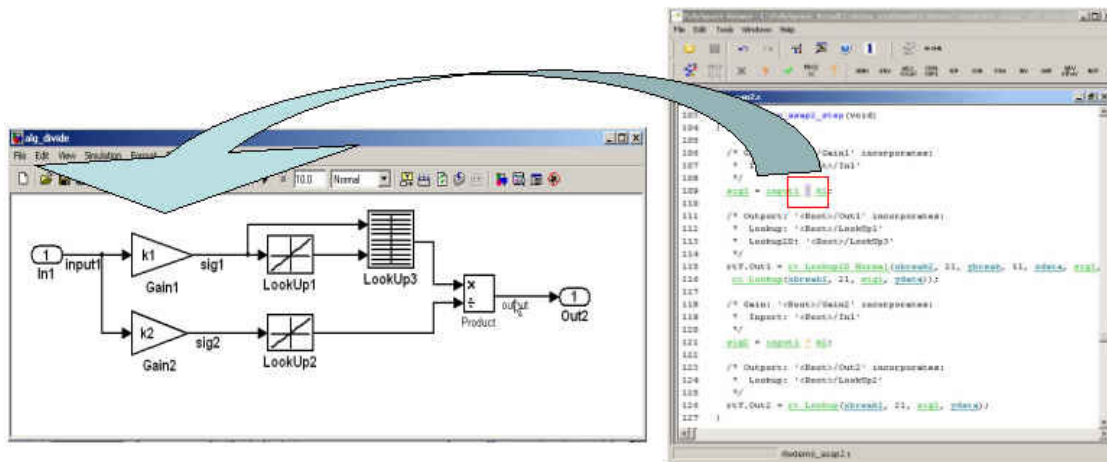


Figure 1: The picture shows the link between Simulink view (left) and PolySpace results view (right). A software developer may look at PolySpace results to identify design areas that may lead to overflow, division by zero and other runtime reliability breaches. The picture shows that the use of gain K1 may lead to an overflow.

Since PolySpace determines all the potential code violations automatically, there is no need for test cases. This has nearly made executing the unit test effortless. The developer now spends time analyzing the test results and not creating the test results. This again is a desirable culture change.

BENEFITS FROM AN INTEGRATED SOLUTION

An example is when Cummins was introducing PolySpace to the product line workflow. Many teams were not familiar with PolySpace and its capabilities. An OEM had experienced intermittent engine shutdowns. Both Cummins and the OEM spent three weeks in a fruitless attempt to isolate the problem. Then

PolySpace was suggested to aid the troubleshooting. PolySpace uncovered an index being decremented in a state flow diagram may go beyond zero resulting in a reset (see an example below).

Since the code execution didn't always decremented beyond zero, the reset appeared random. PolySpace determined root cause, the code was corrected, and the OEM never experienced the issue again.

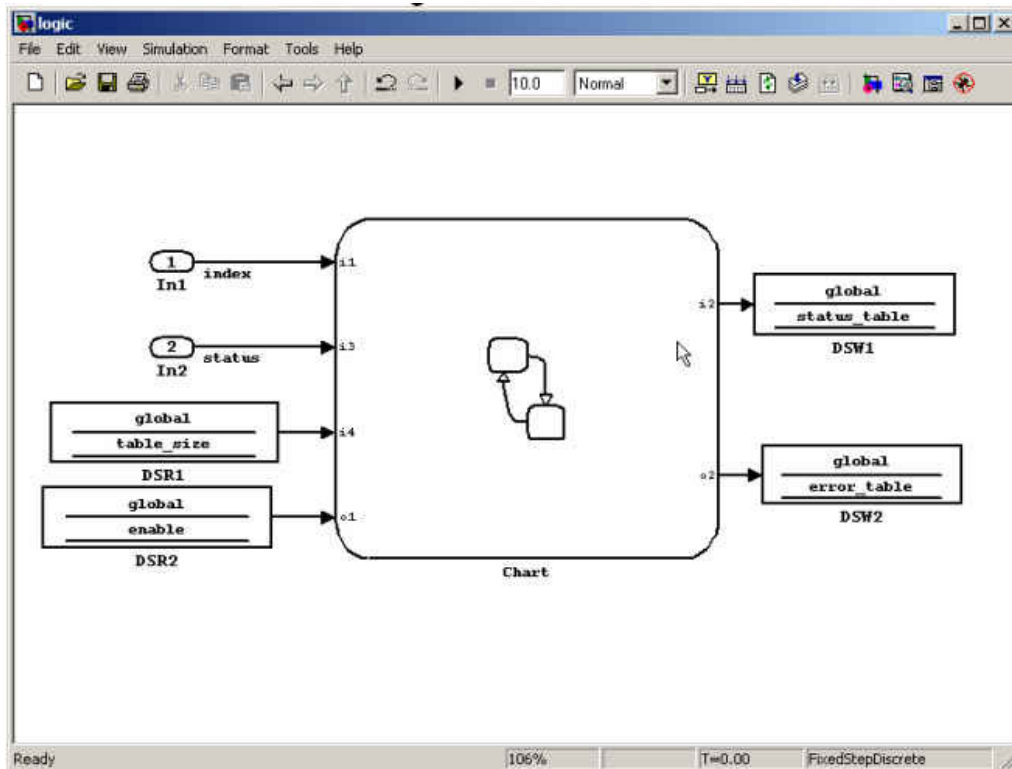


Figure 2: The picture shows a design example involving a state chart used within a block. An operation within the state chart may lead to an out-of-bounds array index. E.g.:

```
if (status == error_type_2) {  
    error_table[index]+1;  
    status_table[index++];  
}
```

Many others have followed the above success, but this time the issues were caught before the system went into the field. There were divide by zero issues that were detected by PolySpace and resolved on development code. Since all these were found and corrected before the software release, the savings to Cummins is conjuncture and certainly an experience gladly avoided.

The synergy between PolySpace and MathWorks products has reinforced awareness that mistakes can be engineered into a product line. Cummins uses The MathWorks tools because of the power of Model-Based Design. As with

powerful design tools everywhere, the engineers can design in unintentional errors that manifest themselves in the underlying software. With PolySpace integrated into the MathWorks toolset, the developer is cognoscente of inherent design constraints.

How is PolySpace applied at a component level?

When self-contained code (the entire system) is executed, PolySpace is cognizant of all the relationships. Anything less than the total system, PolySpace flags potential real time errors (orange), which is how Cummins has chosen to test due to Cummins product line architecture implementation. These orange errors are potentially false indicators of functioning code: type I errors. Again PolySpace takes a conservative position and flags all potential errors; In absents of all the files, more code is potentially in error. So how do you reduce type I errors?

Reducing type I errors is accomplished by including files outside the component at the component interface. This allows PolySpace to be cognizant of both ends of the interface. By including more files, the total operations have increased but the original component under test will have its same total operations. This original total will have shifted some of the orange (potential errors) to the green (good code). Possibly some of the gray (non-executable code) will have moved to the any of the other categories (green, red, or orange). In this way, type 1 errors have been reduced which is a fundamental concept in testing a subset of the entire system.

How about type II errors (failure to detect)? Cummins intentionally seeded 25 different software errors into a component. The placement of the errors determined the detection level of the type 2 errors not the particular type of error. The errors could not be arbitrarily inserted into the code with disregard to sequence of instructions. PolySpace caught all but three errors. The placement of the errors determined the detection level of the type 2 errors. The errors could not be arbitrarily inserted into the code with disregard to sequence of instructions.

All these observations are recorded in the Cummins style guide. This provides the designer with the best practices for the Cummins product line architecture implementation. The unit test issues are corrected in the Simulink diagrams and then generalized to apply across the product line. By abstracting each issue into the style guide, Cummins avoids reoccurring issues and doesn't need to rediscover solutions. The Simulink diagrams are refined and matured.

Cummins developed a script to interface the component version control directly to PolySpace. The developer indicates the component and version that is to be tested. The script assembles all the .c and .h files associated with that component. This eliminates confusion of exactly which versions of files that were tested. In product line architecture, not all product lines are on the main latest

provides an absolute means of the outstanding issues. The trend curve can be fitted to the relevant distribution and predictions can be estimated on completion.

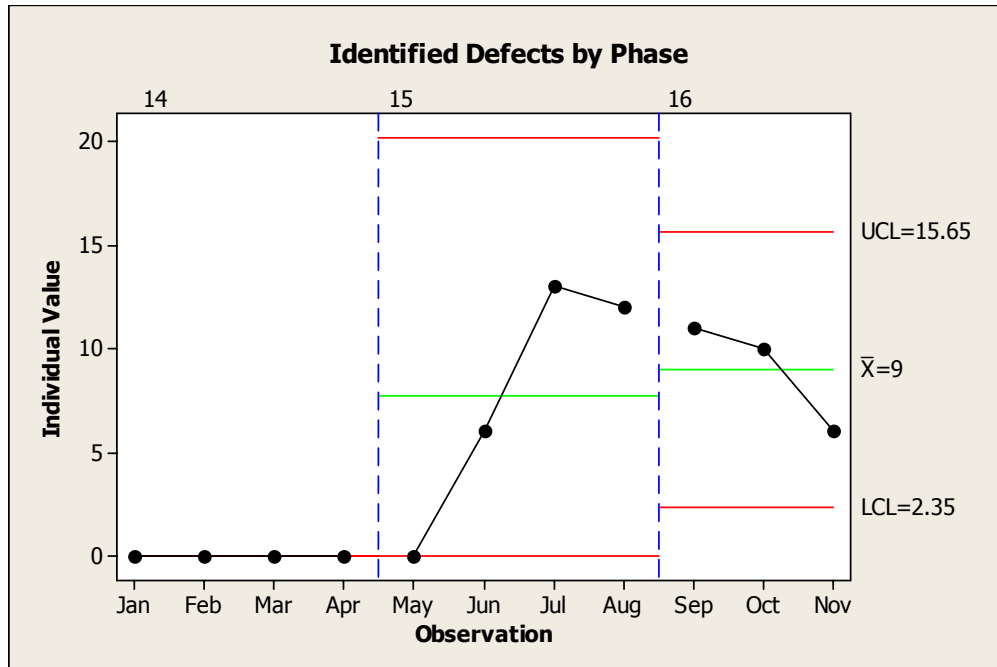


Figure 4: The picture shows how the PolySpace metric evolves over time for a single component. Red lines indicate lower and upper limit or acceptable software reliability metric.

As the chart indicates, the number of defects is going down over time.

CONCLUSION

The paper describes how a workflow that relies on the joint usage of MathWorks and PolySpace products benefits from executable specifications, automatic code generation and automatic code verification.

MathWorks products allow switching from cumbersome code-based maintenance to the maintenance of executable specifications that are easier to read and to understand. By relying on automatic code generation, all errors that relate to specification misinterpretations, hand-written coding, are removed: Automatic code generation faithfully translates executable specifications to C code.

The PolySpace products perform automatic code verification instead of error-prone, costly unit tests, and manual code review. While the PolySpace products may reveal design breaches in Simulink models, they also show which are

correct operations so that the PolySpace results, being exhaustive, are then used to monitor quality over time as a software reliability metric.

Cummins has gained quality improvements through the use of this joint tool set of Mathworks and PolySpace. By correcting the code early in the development cycle, performance engineers are not troubleshooting development engineers' issues. Failure modes are not masked. Improvements are recorded in the style guide to carry forward best practices. Solutions are not reinvented. By only taking into account issues avoided by the component engineer, Cummins conservative estimate is a quarter million dollars saving annually. The savings are much higher if the potential system & performance issues, OEM rework, and customer campaigns are factored. The true higher savings are not known since PolySpace is a proactive tool.