AIAA Modeling and Simulation Technologies Conference and Exhibit
21 - 24 Aug 2006, Keystone, Colorado

AIAA 2006-6257

Automatic Flight Code Generation with Integrated Static Run-Time Error Checking and Code Analysis

Tom Erkkinen[1]
The MathWorks Inc., Novi, MI, 48375

Chris Hote[2]
Polyspace Technologies, Boston, MA, 01801

Model-Based Design with automatic flight code generation is used to develop a variety of aerospace electronic systems including integrated avionics, jet engine controls, and flight control systems. The software deployed in these systems undergoes rigorous verification and validation because the consequences of software failure are severe. New code checking and analysis tools have recently emerged with tight integrations to the modeling and code generation environment provided by Simulink.

I. Introduction

Traditional flight software module development involves paper designs and hand coding followed by verification activities such as code inspections, structural code coverage analysis, and unit/integration test. Many of these activities lack tool automation and involve manual interaction. Thus they are error prone and time consuming. Lack of tool chain integration provides another opportunity for errors to be injected into the software that are often detected late and at high costs to the development process.

A better approach is to use Model-Based Design for creating executable specifications, automatically generating flight code, and performing verification and validation (V&V) activities on the model. Furthermore, recent tool integrations now allow software engineers to easily verify the generated code using static analysis techniques. One such tool integration provides run-time error checking and C code analysis such as MISRA-C[1]. This analysis may be applied very early in the software design process for finding software reliability breaches before functional tests are performed, or applied later for a final sanity check.

Flight software development using Model-Based Design often begins with high-level system requirements. The requirements are allocated to hardware and software and a refinement phase occurs. Eventually a detailed software design model is produced and a V&V analysis and test phase is performed to ensure the model satisfies the known and derived requirements. The test results are also examined to ensure that the software satisfies the requisite structural model coverage.

Requirements traceability between model design components and the higher-level requirements specification is usually involved. Designs specified using state machines and block diagrams can have automated links that support bi-directional traceability to higher-level requirements in popular documentation and requirements management tools. Industry standards such as DO-178B[2] and CMM-I[3] require these high degrees of traceability.

Automated documentation tools help complete the design phase by making it easy to prepare and execute formal and informal requirements and design reviews. These software development reviews are now executed much quicker and easier than before by having rigorously tested models with high degrees of traceability easily accessible.

After satisfying the design phase, the “golden model” is placed into configuration management and code is automatically generated.

---

[1] Manager, Embedded Applications
[2] General Manager.

The code is then assessed for performance, size, and resource consumption. Iterations will often occur to optimize the generated code for a particular hardware platform. Model guidelines used during design can make it easier to get more efficient code during the first iteration. Software integration gradually takes place during the iterations. Integration involves application software and real-time operation system software; external or legacy code integration for some software components such as look-up tables, calibration data, and scaling choices; and low level device drivers for software to hardware interaction.

As implementation and integration winds down, V&V of the generated software takes place. However the sooner V&V efforts occur, the sooner errors are detected and resolved.

With recent technology derived from lessons learned in aerospace programs, it is now possible to perform rigorous analysis of the generated code, such as run time error detection, using advanced static analysis techniques. Furthermore 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 and perform traceability analysis. This paper describes a working example of an integrated environment using Model-Based Design with automatic code generation and advanced code analysis tools.

II. Model-Based Design with Automatic Flight Code Generation

Model-Based Design supports systems and software engineers by providing a common environment for graphical specification and analysis. Models are used to specify system architecture, component interfaces, feedback control, signal processing, state machine logic, and real-time behavior. To satisfy safety requirements, Model-Based Design must address industry standards such as DO-178B and produce the artifacts required by certification authorities. To be used in flight, it must also yield an efficient final executable program that is able to complete its processing task within the time and space allotted by the embedded system hardware.

Model-Based Design development activities discussed in Section II include:

- System Specification
- Software Detailed Design
- Production Code Generation

Other Model-Based Design activities such as bidirectional requirements traceability, model coverage analysis, and report generation were described previously[4] and not repeated here.

A. System Specification

Models are initially used for specifying system functional behavior. A typical system model includes blocks, subsystems, filters, and table lookups in Simulink; state machines, truth tables, and control flow logic in Stateflow; signal processing algorithms written with Embedded MATLAB functions; or hand written code in C (S-Functions).

Large models can be partitioned using model blocks, which allow one model to reference another. Model referencing saves significant simulation and code generation time over the partitioning approach of atomic subsystems in libraries. Model blocks now allow engineers to build and simulate systems approaching a million blocks. Model referencing also supports incremental code generation such that new code is only generated for model blocks whose referenced model has changed. Code is not regenerated for models that did not change. Thus code could be verified just once for each function represented by the model block, saving significant time and effort.

Buses and corresponding bus objects provide lock down interfaces to model block components that may have dozens or even hundreds of interface control description (ICD) data. During code generation, buses can be made non-virtual so that they produce structures (C struct types) in the generated code. With release R2006a, it is now possible to automatically convert atomic subsystems to model blocks as well as to automatically create bus objects as shown in Figure 1. This makes it easy for aerospace engineers to perform design tradeoffs in architecting their system models.

Flight software developers interact with a model using in ports and other input mechanisms. They observe results using scopes and record data by logging it to the workspace. The new signal and scope manager lets you add signal

generators, log signals, and view scopes without adding blocks to models. The “antenna” and “eye glass” icons shown in Figure 1 illustrate that non-intrusive log and scope mechanisms also work for buses.

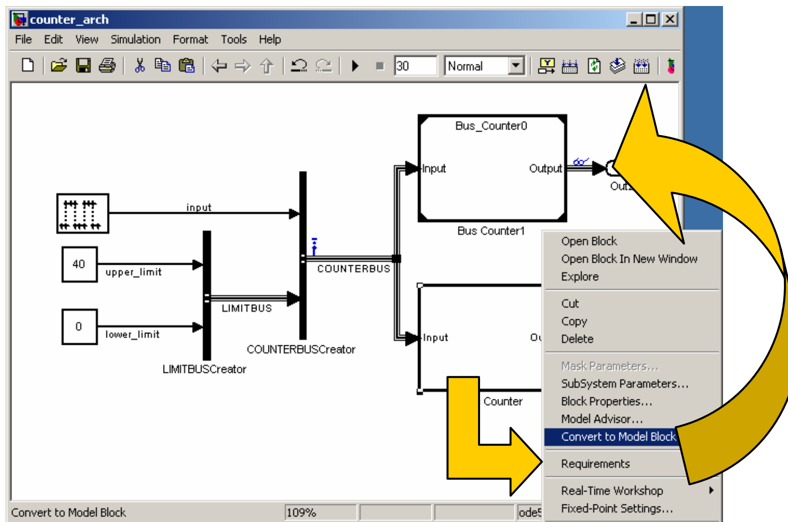


Figure 1: Atomic Subsystems Conversion to Model Blocks.

With R2006a, developers can now model with embedded MATLAB functions in Simulink or Stateflow and produce C code from these functions using Real-Time Workshop Embedded Coder. These functions include fixed- and floating-point logic and are based on a documented subset of the MATLAB language. The subset supported has doubled since the initial release and now also supports advanced MATLAB functions including QR and EIG. In this way, aerospace engineers can develop matrix-based signal processing algorithms, such as Kalman filters for radar applications, with just a few lines of MATLAB code.

B. Software Detailed Design

With Model-Based Design, the model developed by system engineers can be refined and constrained by the software engineers as part of the flight code generation process. Detailed software design often involves single precision data specification, parameter tuning and data sets, and code generation configuration settings for the flight hardware environment. A single model can be used by software engineers and deployed on multiple targets to assess architectural impacts using a data dictionary driven modeling style. To do this, Simulink data and code generation settings can be specified external to the model and loaded into the workspace when needed, as shown in Figure 2. This allows a single model to be easily retargeted for various hardware architectures.

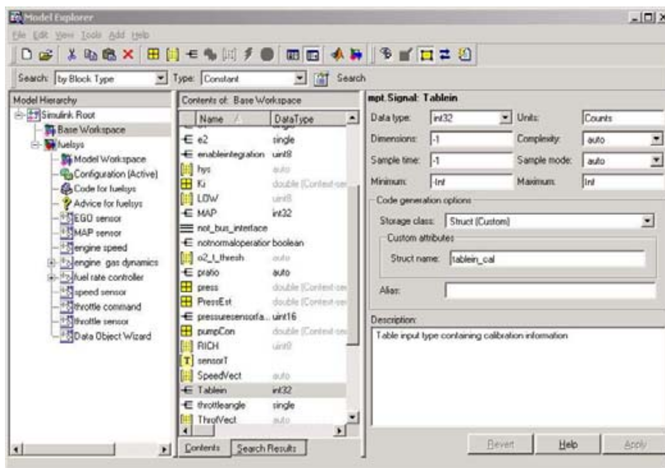
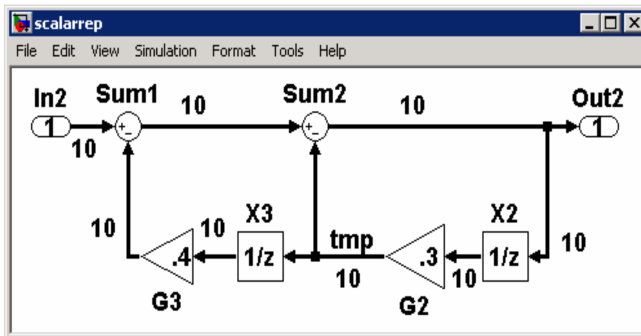


Figure 2. Model Explorer View of Data Dictionary.

C. Production Code Generation

It is time to generate code once the detailed design has been reviewed and verified. As with a C compiler, the code generation process is straightforward. Various optimization settings and user configuration options exist and engineers assess the settings to determine appropriate values. Once done, the configuration setting values may be saved and reused throughout the project as a configuration set and used for multi-target modeling as described earlier for data dictionaries. The settings can be global and affect all aspects of the model, or may be local and used only for particular subsystems.

With R2006a, a number of code generation enhancements have been added that let you generate code that is efficient and easy to integrate with. Figure 3 shows a code efficiency optimization for vector operations. In this case a single for-loop is used to process the vector of 10 elements. Note that the intermediate variables that do not need to be arrays are now just scalars, saving valuable stack space.




```
void scalarrep_step(void)
{
 int32_T i1;
 real_T rtb_tmp;
 real_T rtb_tmp_b;

 for(i1 = 0; i1 < 10; i1++) {
  rtb_tmp = rtDWork.X2_DSTATE[i1] * 0.3;
  rtb_tmp_b = (rtU.In2[i1] - rtDWork.X3_DSTATE[i1] * 0.4) - rtb_tmp;
  rtY.Out2[i1] = rtb_tmp_b;
  rtDWork.X3_DSTATE[i1] = rtb_tmp;
  rtDWork.X2_DSTATE[i1] = rtb_tmp_b;
 }
}
```

Figure 3: Model and code optimized for wide signal operations.

To improve integration, Real-Time Workshop Embedded Coder has added a function export capability. By this, model developers are able to generate code just for function call triggered subsystems. The function code generated lacks scheduling logic and attributes except those needed by the function itself. This allows an external scheduler to easily call the code generated for the functions. A related topic is that a subsystem can now declare their data structures that contain just the data needed to call and use the subsystem. This makes it easy to unit test the code generated for the subsystem without including code or data from the rest of the model. It is also now possible to specify the memory sections or pragmas with which to place the code and data as shown in Figure 4.

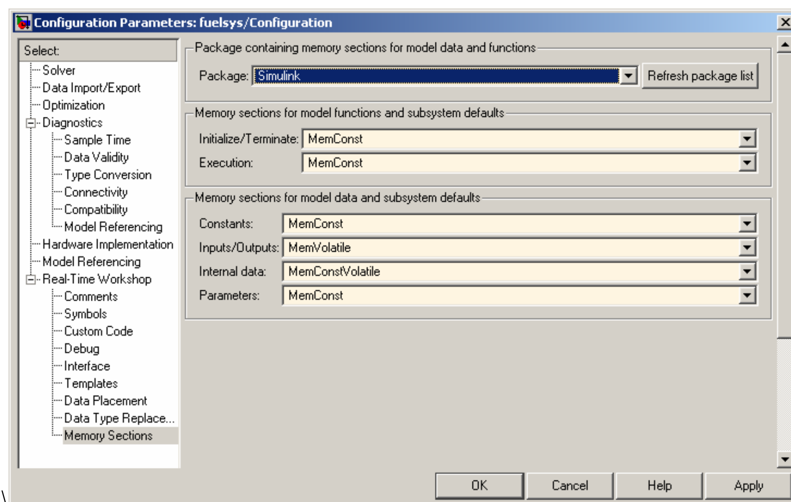


Figure 4: Special memory sections for model and subsystem data and code.

Another area of continued feature growth is template file specification. Templates are one aspect of module packaging features which let developers control and package code files based on production needs or company standards. For example, one can generate data specification files during the code generation process needed for tuning. Once the code generation setting and optional template files are established, code is automatically generated and deployed for the flight environment.

III. Code Analysis Tools

D. Automated Techniques for Flight Code Verification

Traditional source code analyzers focus on error detection by looking at source code. These approaches have been proven very efficient for decades as it leads to early detection of problems: This is to reduce the costs of debugging those problems later when functional tests are performed or when issues raise in the field. However the drawbacks of those mature techniques fall into two different categories:

- Too many false positives: a static analyzer reports an error message when it is not.
- Unknown number of false negatives: a static analyzer does not report anything although there is a bug.

In order to solve those issues, recent static code verification based on abstract interpretation techniques have been applied to detect so-called runtime errors. These errors are also known as dormant faults as they surface for very specific combinations of input data and specific configurations of software applications causing unexplained behaviors or the sending of incorrect commands to actuators. Overflow, underflow, division by zero, buffer overflow, illegal pointer de-referencing, read access to non-initialized data are some examples of runtime errors.

As explained in length in several articles[5,6], static verification by abstract interpretation can handle dynamic properties of programs by solely relying on source code and can exhaustively diagnose the sections of code that may or may not lead to failures. As a result, any code section will necessarily fall into one of the 4 following categories (see figure 5):

- *Reliable: the operation under consideration will never fail because of a runtime error*
- *Incorrect: the operation under consideration will fail*
- *Questionable: the operation under consideration may fail under certain circumstances*
- *Unreachable: the operation under consideration cannot be activated*

```
70        static void Pointer_Arithmetic ()
71        {
72          int tab[100];
73          int i, *p = tab;
74
75          for(i = 0; i < 100; i++, p++)
76            *p = 0;
77
78          if(get_bus_status() > 0)
79          {
80             if(get_oil_pressure() > 0)
81               *p = 5; /* Out of bounds */
82             else
83               i++;
84          }
85
86          i = random_int();
87          if (random_int())  *(p-i) = 10;
88
89          if (0<i && i<=100)
90          { p = p - i;
91            *p = 5;     /* Safe pointer access */
92          }
93        }
```

Figure 5: Static verification exhaustively checks each code section and provides with a detailed diagnostic for each operation that falls into one out of 4 categories: reliable (green), incorrect (red), questionable (orange) and unreachable (gray)

Therefore, static verification by abstract interpretation bridges two important gaps in static analysis techniques:

- False negative removal: indeed, every code section of code is verified and gets a diagnostic as far as runtime error is concerned.
- False positive reduction: By applying inter-procedural analysis, aliases analysis and analysis of control structures, error detection now relates to the operational semantic of the code not only on syntax and static aspects of it.

The PolySpace company successfully pioneered static verification techniques few years ago and developed products for the embedded systems industry where software reliability is at stake. Products are available for MISRA-C:2004, C, C++ and Ada programs and aim at pinpointing errors in code prior to functional testing and at providing an actual software reliability measurement of production code (source code reliability and error avoidance metric, see figure 6).

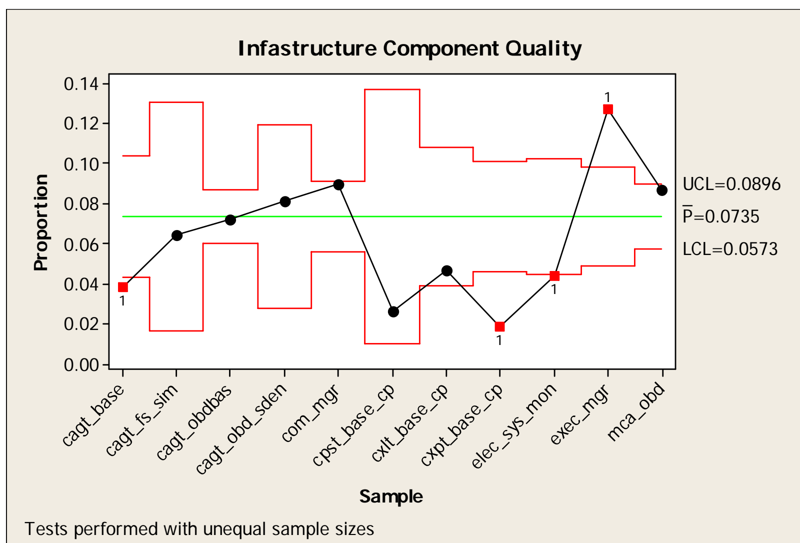


Figure 6: PolySpace diagnostics are a good indicator of software reliability level. The figure shows an example[7] where the PolySpace diagnostics are displayed from a collection of C files. Average and standard deviation can also be displayed to quickly highlight components in which software quality can be improved. PolySpace diagnostics also indicate the categories of problems that first need to be investigated (overflow, out-of-bounds array index, read access to non-initialized data, division by zero …).

The following paragraphs elaborate on how exhaustiveness of PolySpace diagnostics is used in the context of automatic flight code generation as to find design flaws due to scaling choices, implementation constrains or inappropriate design before those errors disrupt functional tests and simulations. As a result, PolySpace products can be used on auto-generated code to significantly reduce time spent in code verification and code coverage activities.

E. Integration of Simulink and Polyspace

With the production of automatically generated code, developers need a means to quickly and efficiently test, analyze and debug the application. PolySpace provides run time error analysis of the code and has recently worked with MathWorks to integrate the code analysis of automatically generated code with the corresponding Simulink model.

Figure 5 shows the tool chain integration.

- On the left side of the figure is a Simulink subsystem.
- On the right side is the generated code displayed with PolySpace diagnostics.
- The diagnostics indicate that an overflow may occur in the code (conversion to “int16”).

The orange code section (squared in red) in Figure 5 indicates a possible design breach and is linked to an operator in the Simulink model. The green sections show operations are reliable under all circumstances. By looking at the Simulink model, it appears the “k1” gain operator needs to be assessed. The breach is caused by the fact an external data is directly connected to the gain. Depending on value of the data an overflow may happen and therefore a protection should be added here (e.g.; Limitor operator).

Note that the PolySpace analysis is activated through an invocation done directly in Simulink.

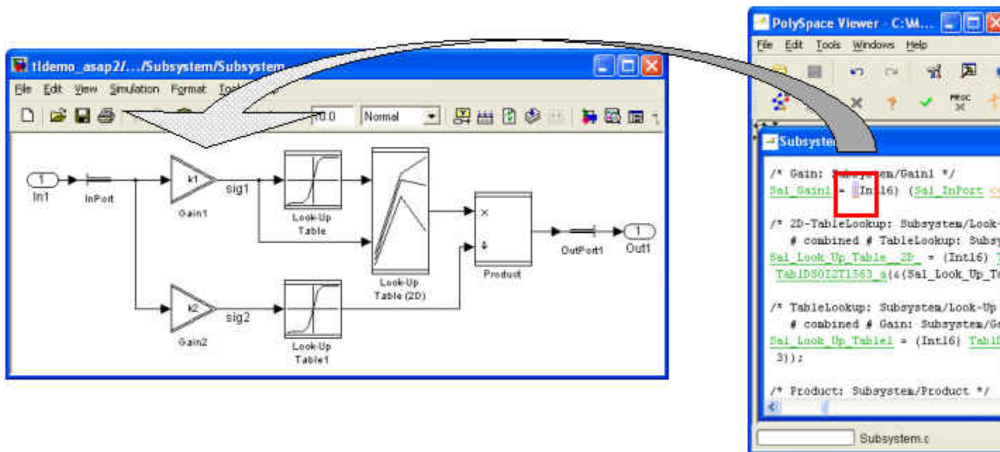


Figure 5: The picture shows the link between Simulink view (left) and PolySpace results view (right). A software developer may look at PolySpace orange code sections to identify questionable design areas including overflow, zero divide and other runtime reliability breaches.

In addition to overflow and divide by zero, Polyspace analysis can also find a number of other potential design errors. In one case, a company experienced intermittent engine shutdowns. They spent three weeks in a fruitless attempt to isolate the problem. PolySpace tools were used 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 did not always decremented beyond zero, the reset appeared random. PolySpace determined root cause, the code was corrected, and the OEM never experienced the issue again[7].

See figure 6 for a possible array out of bounds example. In this case, the code may be generated as follows:

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

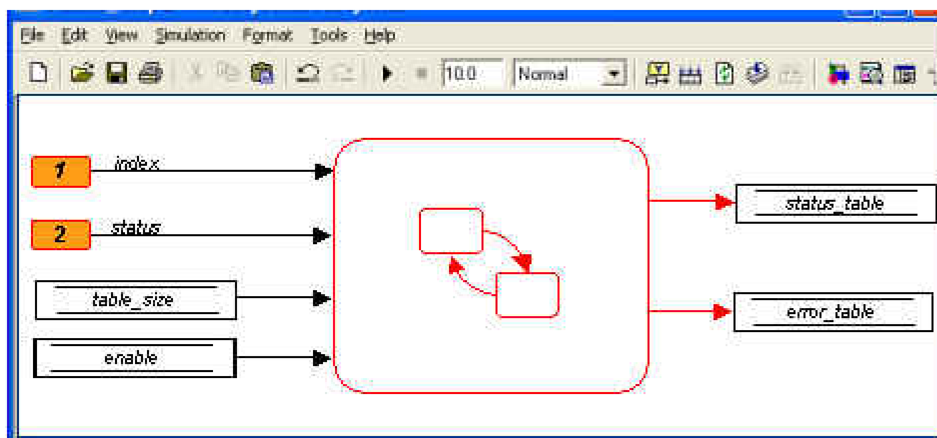


Figure 6. An operation within the state chart may lead to an out-of-bounds array index. E.g.:

The synergy between PolySpace and MathWorks products has reinforced awareness that mistakes can be engineered into products. As with any tool or development approach, engineers can design in unintentional errors

that manifest themselves in the underlying software. With PolySpace integrated into the Model-Based Design toolset, the developer is much more cognoscente of inherent design constraints.

IV. Conclusion

For flight software, it is important to perform rigorous analysis of the flight code whether generated by hand or using automatic code generation. Run time error detection and advanced static analysis techniques aid the code analysis. Technology exists that supports links for the code analysis tool back to the original model and applies to different levels of the model, including subsystems and blocks. This makes it easy to resolve issues and perform traceability analysis.

References

[1]"Guidelines for the Use of the C Language in Critical Systems", ISBN 0 9524156 2 3 (paperback), ISBN 0 9524156 4 X (PDF), October 2004. http://www.misra.org.uk.
[2]"Software considerations in airborne systems and equipment certification", RTCA/DO-178B, RTCA Inc., Dec. 1992.
[3]CMM-I, http://www.sei.cmu.edu/cmmi/
[4]Erkkinen, T. J., "High Integrity Production Code Generation," AIAA Guidance, Navigation, and Control Conference, AIAA, Washington, DC, 2003.
[5]Deutsch, A., "Static Verification of Dynamic Properties," http://www.polyspace.com/white_papers.htm.
[6]Hote., C. "Advanced Software Static Analysis Techniques That Provide New Opportunities for Reducing Debugging Costs and for Streamlining Functional Tests," http://www.polyspace.com/white_papers.htm.
[7]Harmon, R., Hote, C., "Automatic Engine Control Code Generation with Integrated Automatic Static Code Verification," International Automotive Conference 2006, http://www.polyspace.com/white_papers.htm.