Using MathWorks' Simulink® and Real-Time Workshop® Code Generator to Produce Attitude Control Test and Flight Code

Mark Andrew Salada, Associate Staff
Johns Hopkins Applied Physics Laboratory
(443) 778-7267 (Baltimore)
(240) 228-7267 (Washington, D.C.)
Mark.Salada@jhuapl.edu

Wayne Dellinger, Ph.D., Senior Staff
Johns Hopkins Applied Physics Laboratory
(443) 778-7565 (Baltimore)
(240) 228-7565 (Washington, D.C.)
Wayne.Dellinger@jhuapl.edu

Abstract

This paper describes the use of a commercial product, MathWorks' Real-Time Workshop® (RTW), to generate actual flight code for NASA's Thermosphere, Ionosphere, Mesosphere Energetics and Dynamics (TIMED) mission. The Johns Hopkins University Applied Physics Laboratory is handling the design and construction of this satellite for NASA. As TIMED is scheduled to launch in May of the year 2000, software development for both ground and flight systems are well on their way. However, based on experience from previous APL missions such as Midcourse Space Experiment (MSX) and the Near Earth Asteroid Rendezvous (NEAR), the designers of the attitude estimation and control system desire a more streamlined approach for analysts to incorporate their algorithms into flight code. Specifically, the attitude control designers want an easier and quicker iteration capability during integration and test that somehow includes their principle development environment, Simulink®. One of the problems is that complete attitude simulations in the Simulink models include both flight and non-flight elements. With a significant initial effort, RTW now separates flight code from the non-flight code, incorporating changes directly from Simulink instead of editing the code after the fact. RTW first converts the Simulink inner workings into a single, "all-knowing" file. The Target Language Compiler™ (TLC) then uses this file to convert the information into actual code. Simulink's RTW product comes complete with "canned" TLC configuration files that control the generated C code. By editing these configuration files, analysts are able to perform complete estimation and control simulations in Simulink, and then with the click of a button produce code that can be directly compiled and linked onto flight systems. This ease has one caveat, however. By empowering the analysts to generate their own code, they also inherit the class of problems associated with real-time embedded systems, such as concerns with time and space efficiency.

Introduction

The Applied Physics Laboratory is progressing on work for the Thermosphere, Ionosphere, Mesosphere Energetics and Dynamics (TIMED) mission scheduled for launch in the year 2000. The mission distributes the science team among university

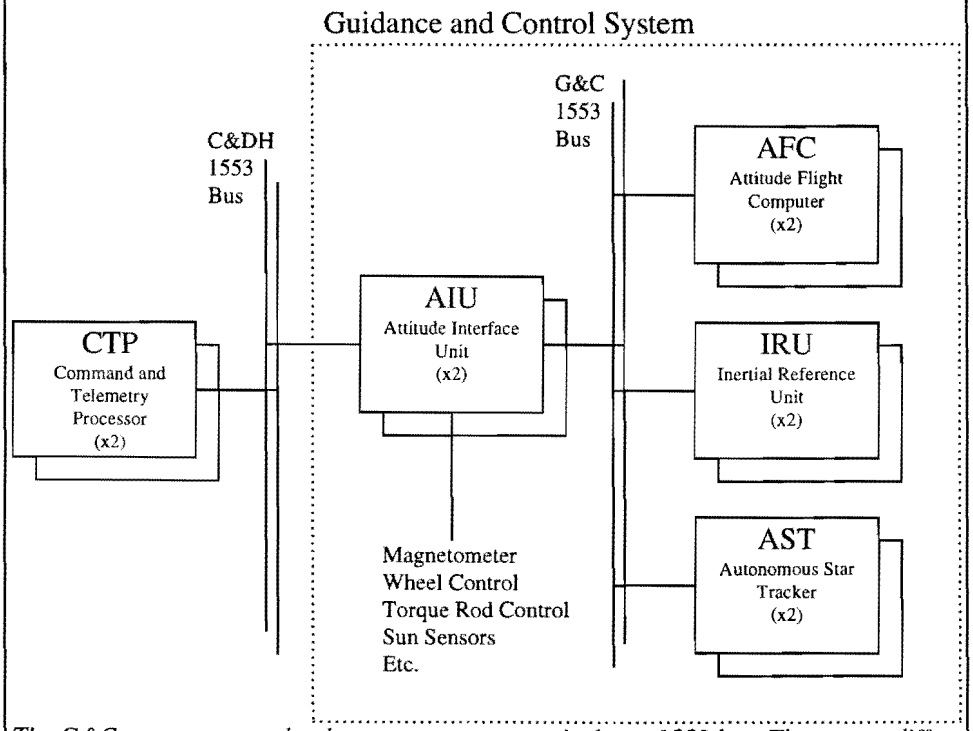
partners, with most instruments developed and operated remotely. APL's principal responsibility is for the spacecraft itself. The satellite mainly consists of a high fidelity pointing platform with minimal maneuvering to accommodate multiple, concurrent science activities, mostly Earth observing. An on-board GPS system maintains ephemeral information which the attitude control system uses closed-loop. This way, TIMED mission operations have the potential to run autonomously, with minimal interference from the ground, truly spreading the satellite operations to the distributed base of science users. With this simple set of attitude control requirements, the APL's TIMED attitude and control team are using the opportunity to explore a new software engineering trend in industry: auto-code generators. The motivation for auto-code comes from the need to reduce the number of people between algorithm development and flight code (i.e., to streamline the process). On previous missions, such as APL's Midcourse Space Experiment (MSX) and Near Earth Asteroid Rendezvous (NEAR), analysts produced the attitude control flight code that flight programmers eventually ingested in the final code. Although both missions proved the process successful, neither mission produced flight code that could be easily re-used on subsequent missions. Eager to establish a more controllable process, and to reduce the cost of starting from scratch each mission, APL is attempting to streamline the attitude control flight code process with new commercial auto-code tools. The new approach attempts to eliminate repeated efforts in testing and is geared towards establishing a unified attitude flight code front from one program to the next.

The opportunity to take advantage of an auto-code generator has existed in industry for many years. Use of code generators, however, remains small. This is probably due to basic inadequacies in existing products, such as the ability to generate code piecemeal, and then re-ingest into the simulation. All existing code generator tools require a heavy initial effort, especially when considering the product for flight code. The TIMED attitude team decided to invest in MATLAB and RTW for two reasons. First of all, it is the same environment used for algorithm development. The second and empowering feature in MATLAB and Simulink is the open architecture which allows full end-user control of Simulink's code generating engine, RTW. With a one-time effort, MATLAB users can modify the code generated from Simulink to fit any flight code platform, and any non-flight platform for that matter. There are about a half dozen important issues with APL's RTW modifications. This paper describes the two most important changes: the re-organization of the Simulink blocks, and the removal of a key, but cumbersome data structure. The TIMED attitude team is also careful to establish the performance and test of the auto-code. The final auto-code product will reside on three platforms, two of which are actual flight platforms.

Development Environment

It has long been the preference of attitude control analysts to use a graphical tool for development. ISI's MATRIX$_X$® and MathWorks' Simulink are two such tools. APL has used both in the past, and has until recently favored ISI's due to its ability to generate Ada code. The TIMED mission attitude developers, however, use Simulink due to its ability to generate ANSI C code. Development for the TIMED mission started before the release of MATLAB 5.2, and therefore APL continues to use version 5.1 for all attitude control development. This is not a version freeze in any sense, and ana-

Figure 1:



The G&C system uses redundant components attached to a 1553 bus. The system differs only slightly from previous APL missions. The AIU is responsible for all bus management, and sensor/actuator interface. The AFC provides all guidance and control during operational modes.

lysts intend to upgrade as soon as the ramifications for the auto-code generator, an add-on to MATLAB, are apparent. Additionally, analysts use a set of SGI and HP machines running UNIX as a shared development environment. There is a particular snag with such a diverse development environment because analysts need to maintain separate versions of '.mex' files for S-functions. (There is more detail on S-Functions below.) The new SGI 64-bit machines require another set of '.mex' functions, bringing the total configuration management mess multiplier to three. With a finite and relatively small set of '.mex' functions to date, the multi-platform attitude control development schedule remains manageable.

Target Environment

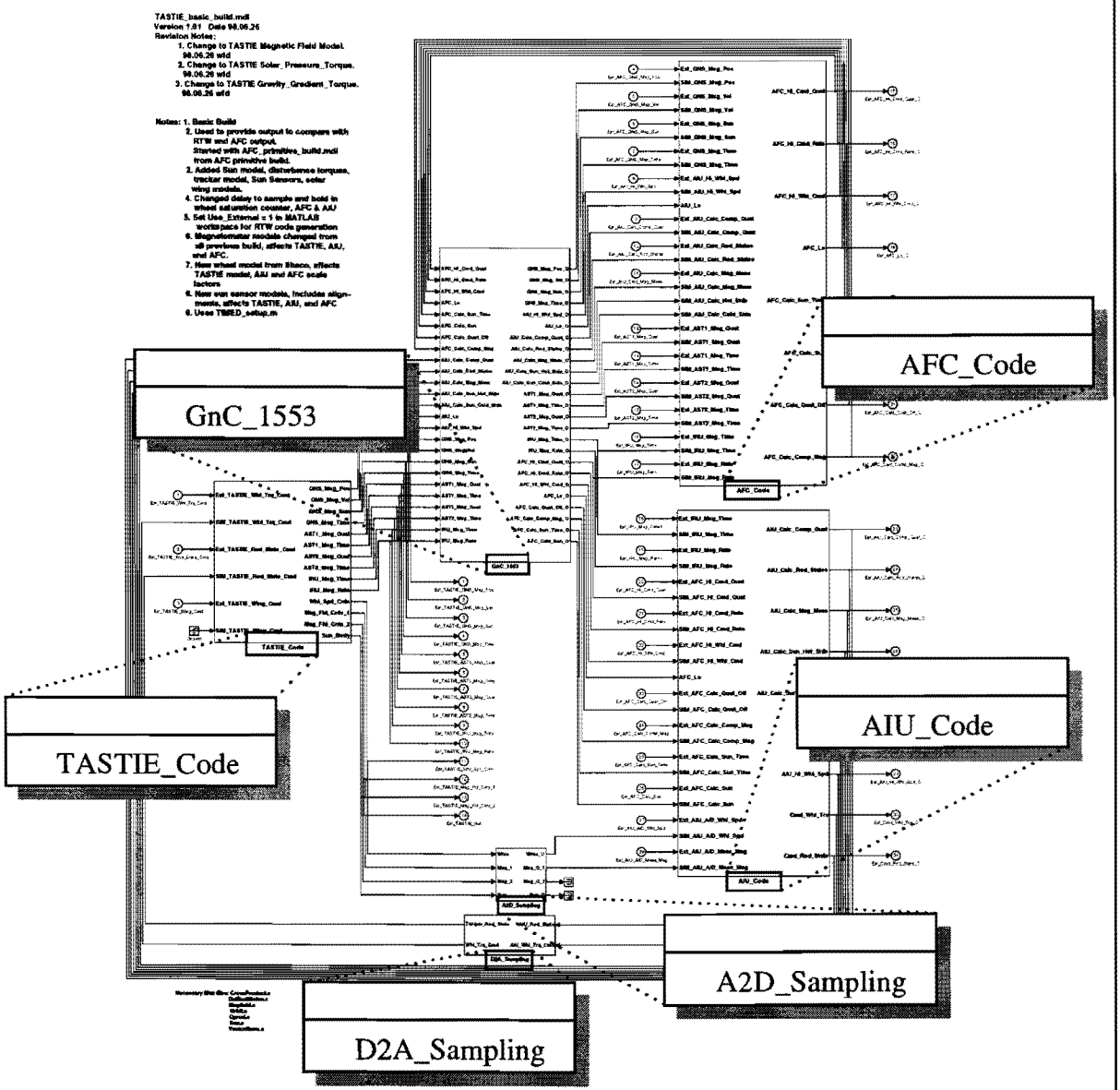
TIMED is a three-axis stabilized, nadir pointing platform with tight constraints about every axis (0.03° of knowledge, 0.5° for control). The attitude control suite includes a magnetometer, three torque rods, reaction wheels, two star trackers, a redundant inertial reference unit, sun sensors, and finally a set of two processors, each with a redundant twin (See Figure 1). The Attitude Interface Unit (AIU) is a 16-bit RTX 2010 processor primarily responsible for sensor interface, bus management, and remedial attitude control (safing). The Attitude Flight Computer (AFC) is responsible for all attitude control, except safing, and utilizes the Mongoose V 32-bit processor. APL has considerable experience with the AIU/AFC attitude control combination, borrowing lessons learned from both the MSX and NEAR programs. A standard double redundant 1553 bus connects the two processors to each other, along with the star trackers and inertial reference units (IRU). A separate 1553 bus serves as the interface to the command and data handling system through the AIU.

By far, the constraining element of the system is the AIU processor, the RTX 2010. With 128k of memory, and 128k of PROM, the decision to use auto-generated code came with considerable wariness. In fact, system developers at APL did not finalize usage of the auto-generated code until

analysts produced a primitive set of AIU attitude code, and performed a baseline performance test on the engineering model. More detail concerning the performance and constraints imposed by the processor is below. Confidence in the auto-code approach continues to grow.

The third platform for the auto-code is a non-flight platform called TASTIE. TASTIE stands for the TIMED Attitude Simulation and Test Integration Environment. It is the primary testbed for the satellite guidance and control system. All of the environmental models and command and data handling stimulus reside on a separate component rack, acting as the rest of the satellite and ground system to the guidance system. The main processor runs OS/9 on a Motorola 32-bit 68060 VME backplane. The memory and storage for this environment

Figure 2:



This is the root-level page for the TIMED Simulink simulation. (Apologies for the micro-font) Each user-defined subsystem corresponds to actual spacecraft hardware. This allows for tidy grouping of generated code, and consequent separation of flight and non-flight software from the same simulation.

dwarf the flight platforms, and therefore TASTIE provides the most immediate validation of the auto-code approach.

A Virtual Satellite

The first auto-code misnomer to dissipate is the myth that auto-code generators actually auto-generate code. (There is no magic involved, black or otherwise.) Auto-code generators are simply text manipulators that connect the signals and logical constructs usually specified through an alternate means (e.g., a graphical interface like Simulink instead of an editor). Auto-code generators do no more than what they are instructed to do, and they have the added benefit of doing it quickly and reliably. There are no up-front savings using auto-code generators. In fact, there is a greater than normal initial investment. The savings come by means of down-the-road repeatability, speed, and code integrity.

The attitude control team chose to carefully construct the Simulink simulation to be as "life-like" as possible. In effect, APL created a virtual satellite in Simulink with actual subsystems for satellite hardware items such at the processors and the bus. Figure 2 shows the top-level page in Simulink for the TIMED model. The left-hand side of the diagram holds the command emulator, and all environmental effects for the simulation. The right-hand side represents the spacecraft elements. Note that there are separate blocks for each flight processor. Furthermore, the system models actual spacecraft elements such as the 1553 communication bus, and digital and analog samplers. The reason APL stresses this representation so heavily is that it becomes the primary influence on the way the generated code separates into manageable pieces and compiles.

TLC Script Block Re-Organization

As mentioned before, the empowering element of MathWorks' Simulink is the open architecture. An open architecture describes a commercial product that allows its users to modify the way certain elements function without additional support from the source. The entire code generator in Simulink, also called the Real-Time Workshop (RTW), controls its final code product through a separate scripting language native to the Target Language Compiler (TLC). As delivered from MathWorks, the RTW comes complete with default, or "canned," TLC scripts that produce code. These scripts complete the two step process that turns Simulink block diagrams into a small set of source files.

The first step in the auto-code process creates a 'core' file containing all the block diagram connectivity and organization information. The file has the suffix '.rtw.' MathWorks organizes Simulink information in two basic ways. The first and most basic information element is the block. A block contains anything from a straightforward vector sum or multiply, to a complete DSP function such as a fast Fourier transform (FFT). The second form of information groups these blocks into systems and subsystems. Therefore, the '.rtw' file contains all connection information between blocks, and then between systems and subsystems. (The only difference between systems and subsystems is that user grouped blocks become subsystems, while blocks grouped necessarily, internal to Simulink, become systems. A more detailed explanation of Simulink systems is beyond the scope of this paper.) (See Figure 3.) It is not possible to change the way Simulink generates the ASCII '.rtw' information file. However, it is possible to completely control the way RTW

generates source files from this point onward via TLC scripts.

The second and final step in the code generation process is the invocation of a series of TLC scripts. TLC scripts are a language on their own with full access to all information in the '.rtw' file. Unfortunately, the default organization of blocks, systems, and subsystems does not accommodate the way the TIMED attitude team intends to utilize the code generator. The default organization groups all blocks based on internal systems, mostly dependent on sample rate, completely independent of the user-defined subsystems (See Figure 4). Since APL intends to allow the user to specify which blocks, or groups of blocks, reside on an actual system as embedded code, the default organization is clearly insufficient. Therefore, grouping the generated code from blocks grouped by *subsystem* is the major change made to the default TLC files. Since the '.rtw' information file organizes blocks by the internally-defined systems, there is a natural elegance to the default TLC scripts that do the same. Changing the organization, as we were warned by the MathWorks distributors and support staff, was not trivial.

Figure 3:

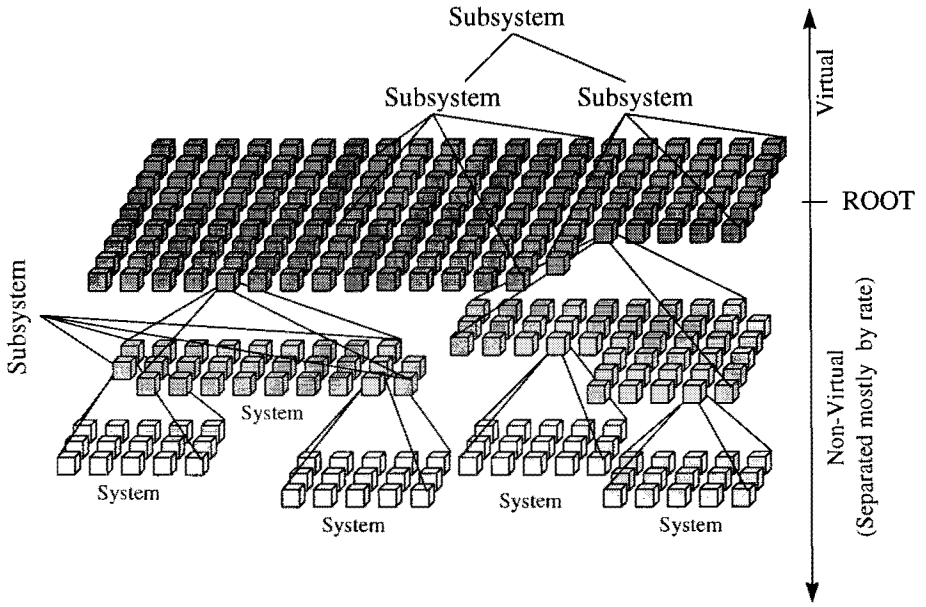
```
CompiledModel {
  Name                    "timed"
  Version                 "2.09 (May, 6 1997)"
  GeneratedOn             "Thu Jul  9 19:12:33 1998"
  Solver                  ode4
  SolverType              FixedStep
  StartTime               0
  StopTime                10000
  FixedStepOpts {
    FixedStep               0.005
    TID01EQ                 1
  }
  DataLoggingOpts {
    LogT                    ""
    LogX                    ""
    LogY                    "yout"
    LogYNCols               [106]
    LogXFinal               ""
    MaxRows                 0
    Decimation              100
  }
  NumModelInputs          83
  NumModelOutputs         106
  NumNonVirtBlocksInModel 1636
  DirectFeedthrough       yes
  NumContStates           7
  NumDiscStates           65
  NumModes                10
  ZCFindingDisabled       yes
  NumNonsampledZCs        0
  NumZCEvents             2
  NumRWork                19
  NumIWork                12
  NumPWork                0
  NumDataStoreElements    4
  NumBlockSignals         2004
  m
```

Figure 3 is an excerpt from the TIMED '.rtw' file as generated by the RTW. This ASCII file acts as a global data set for the Target Language Compiler. The TLC scripts use the field names in this file to generate code, never basing execution on hard-coded numbers. Therefore, only a one-time modification of the TLC scripts is necessary. The file organizes all information by the Simulink-internal systems.

The first step in designing the re-organization was deciding to separate the blocks based on subsystems resident *only on the root page* (pictured above in Figure 2). There are any number of subsystems hidden beneath other subsystems, and throughout the model. Thankfully, the root page has a special status in the .'rtw' file, and therefore it was relatively easy to identify each subsystem. The new implementation that searches the root page has a caveat: Now *only* subsystems may reside on the root page. This means that there can be no stand-alone blocks floating around on the top page, or otherwise the new organization gets confused.

The next step to the re-organization is to scan beneath all of the root-level subsystems and identify all the blocks underneath each. Since the default organization of the '.rtw' file does not explicitly recognize subsystems, this scan process is very cumbersome. In fact, it is merely an iterative

Figure 4:



All blocks in Simulink group into internally defined Systems. The systems, pictured here as horizontal levels of blocks, are primarily defined by sample rate. The 'root' level of blocks, for example, all execute at the fundamental sample rate. The user-defined Subsystems sometimes group blocks within systems, and often across systems. The major change to the TLC scripts is to parse the generated code by subsystem, *not by system.*

loop (unfortunately not recursive) that handles an arbitrarily chosen 'depth' of ten subsystems. This means that when an analyst cascades the subsystems beneath the root page, he or she may only have nine additional levels of subsystems. This constraint does not limit the current TIMED model (with only five levels to date), and can easily expand to another arbitrary limit if necessary.

The third and final step in organizing the blocks by subsystem turned out to be the most complicated. With a set of blocks for each subsystem concisely stored as an array of block numbers, the first thought was to allow the code generator to process each block as it does normally, one at a time, for each subsystem. Unfortunately, the resulting code did not produce simulation results that were identical to the default TLC scripts. It turns out that by separating the blocks into subsystems, some block sequences lost their proper execution order, and therefore destroyed the integrity of the model. Imagine a set of blocks, numbering one to ten in a root-level subsystem. Recall that the blocks are originally organized by system, and this particular subsystem has only one system within it. Then each of the blocks should execute, in order, from one to ten. Now imagine an analyst that intends to group the blocks within the subsystem differently, say into two further subsystems. He or she groups one through five in subsystem A, and six through ten in subsystem B. The model will execute perfectly if, in every case, subsystem A generates code before subsystem B. However, in some common circumstances, the new TLC scripts generate code for subsystem B prior to subsystem A, laying the source lines of code into the files out of order. In most cases, the resulting code contains erroneous signal propagation. Validation becomes impossible because the generated source code outputs will most likely not match the original's models outputs. The example of subsystem A and B is very simplistic, and the code generation block order problem becomes enormous with such complicated models as the TIMED simulation.

How then to guarantee the proper block order for each root level subsystem? The answer is in the block numbering itself. As long as the TLC scripts sort the blocks, separated by root-level subsystem, by num-

ber in ascending order, the code signal integrity maintains itself within a subsystem. Finally, after separating the blocks by root-level subsystem, the new TLC scripts sort the block lists using a standard heap-sort algorithm[i]. Then the scripts feed the block lists to the code generator to produce the code.

Once at the code generator, all changes implement a simple goal: Do not modify the way the default scripts generate the final product (the source code) as much as possible. In other words, the attitude team preferred non-destructive additions to the files as opposed to a slash and burn technique. To this end, the team decided to separate code for the root-level subsystems in the final code using pre-compiler '#ifdef' statements, rather than moving and re-organizing the source lines of code. (The final source code is in ANSI C, where surrounding lines of code with pre-compiler statements, such as #ifdef statements, instructs the compiler to conditionally include or exclude lines of high-level code.) This way, all TLC script additions comprise mainly of inserting pre-compiler statements before and after each block list passes through the generator (See Figure 5). At no point do the new TLC scripts change the way the code generator handles a given block. The only change is the order of the blocks passed to the generator.

Figure 5:

```
  /* Sample hit for TID=2 */
  if(ssIsSampleHit(rtS, 2, tid)) {
   /*  DiscreteIntegrator Block: <S107>/Discrete-Time Integrator */
   rtX.d.s107_Discrete_Time_Integrator[0] = rtX.d.s107_Discrete_Time_Integrator[0] +
0.04 * rtB.s103_Switch[0];
   rtX.d.s107_Discrete_Time_Integrator[1] = rtX.d.s107_Discrete_Time_Integrator[1] +
0.04 * rtB.s103_Switch[1];
   rtX.d.s107_Discrete_Time_Integrator[2] = rtX.d.s107_Discrete_Time_Integrator[2] +
0.04 * rtB.s103_Switch[2];
  }

#endif                          /* AIU_Code */

#ifdef D2A_Sampling
#endif                          /* D2A_Sampling */

#ifdef GnC_1553
#endif                          /* GnC_1553 */

#ifdef TASTIE_Code
  /* Sample hit for TID=4 */
  if(ssIsSampleHit(rtS, 4, tid)) {
   /* UnitDelay Block: <S121>/dt_orbit  */
   {
    int_T i1;
    real_T *u0 = &rtB.s121_S_Function[0];
    real_T *xd = &rtX.d.s121_dt_orbit_[0];

    for(i1 = 0; i1 < 6; i1++) {
     xd[i1] = u0[i1];
    }
   }

   /* UnitDelay Block: <S121>/dt_orbit */
   rtX.d.s121_dt_orbit = rtB.s121_Constant7;
  }
```

This is an excerpt from the generated source file timed.c. The 'raised' sections highlight the new separation by subsystem. Each set of code is surrounded by pre-compiler statements. Therefore all platform engineers use the same source file, timed.c, but with different compilation flags. In some cases, there are no source lines of code associated with a subsystem, like the D2A_Sampling subsystem. This approach allows for later inclusion of code for those subsystems once the real hardware dynamics are better understood.

APL was very successful with this approach. The basic architecture still resembles the default TLC script architecture, with the final product conditionally dependent on the user-defined root-level subsystems. The first target platform, TASTIE, validated our efforts with a relatively smooth integration into operation.

The Removal of Simstruc{}

As mentioned before, there are about a half-dozen important issues with APL's modification of RTW. The block reorganization is the most important change to the TLC scripts. The other most important change to RTW, this time outside of the TLC scripts, enabled the usage of the auto-generated code on the most constraining platform, the RTX 2010 processor. The internal data structure organization of the RTW generated code is very sophisticated. The RTW developers at MathWorks have done an amazing job of implementing a generic code architecture to handle any Simulink block diagram. This generic approach utilizes a single, global structure for each model that coordinates everything from pointers to local data, to references to external functions for the integrators. RTW calls this structure simstruc{}. This structure is very elegant, minimizing recompilation of external source files, and allowing for expansion both in space and function.

With a successful demonstration of the auto-code approach on the TASTIE environment, the next platform seemed a mere exercise. We were remiss to ever believe that. As it turned out, any attempt to simply compile the generated code with the RTX 2010 toolset had the unnerving result of hanging the compiler. There were two significant problems. The first problem was in size. The fields within the data structures declared by RTW were simply too numerous for the compiler symbol table. The second problem was the inability for the compiler toolset to handle forward-referencing structures. We handled the first problem readily by reducing the number of fields declared by RTW with the convenient S-Function hook in Simulink. S-Functions are simply blocks in Simulink where the user can specify a function call. An S-Function compiles from '.mex' files created within MATLAB, and allows for inclusion of source code exterior to Simulink. For example, APL is re-using actual MSX and NEAR flight code for certain functions by hooking their calls and underlying code modules through S-Functions. This approach has the added benefit of reducing the number of input and output signals, and also associated parameters for a given set of blocks. This solved the symbol table overflow problem because the number of signals associated with a block is directly related to the number of fields declared in the RTW data structures. S-Functions provide a means to gracefully scale back the amount of new code actually generated by RTW.

The other compiler problem, with the forward-referencing structure hang, did not have such a graceful solution. In fact, since the RTX compiler tool set was already frozen for development, APL had to either eliminate the problem or abandon auto-code on the AIU. (This compiler problem may have been eliminated in a more recent version of the toolset.) The TIMED attitude team chose to eliminate the problem by eliminating the code dependence on simstruc{}, the only forward-referencing structure in RTW. To retain as much homogeneity as possible, the attitude team removed simstruc{} for all TIMED platforms, not just for the AIU.

To say the least, this is a profound change to the RTW generated code, even though the change only affects the source library files included with the RTW toolbox. The change does not affect the TLC scripts, or the '.rtw' information file. Without exploring the details of the change, there are four things to mention about the elimination of simstruc{}. The elimination of simstruc{} requires a custom modification to any integrator utilities in Simulink. (The modifications arise because simstruc{} contained all

storage structure references for continuous states.) When the user (analyst) selects an integrator in Simulink, he or she has the choice of fixed or variable-step, 0^{th} through 5^{th}-order, and a myriad of techniques, all selectable by pull-down menu. To date, the attitude team has adapted only one integrator for the removal of simstruc{}: the Runge-Kutta 4^{th}-order fixed step integrator. Therefore, without further custom work on other integrators, the TIMED attitude control developers have to use just one type of integrator.

The second thing to mention about the removal of simstruc{} is the loss of nearly all RTW "implied integrity" and support. Basically, the level at which APL modified the delivered source libraries, and TLC scripts, removes any possibility of support from MathWorks, and raises the question of reliability. Can APL depend on the RTW commercial quality now that the final product looks so different from MathWorks' original delivery? The answer is no. Therefore, the TIMED attitude team established a rigorous test, verification and validation plan for the modified product, relying only partly on the commercial integrity. The testing section below describes APL's approach to testing and validating the auto-code.

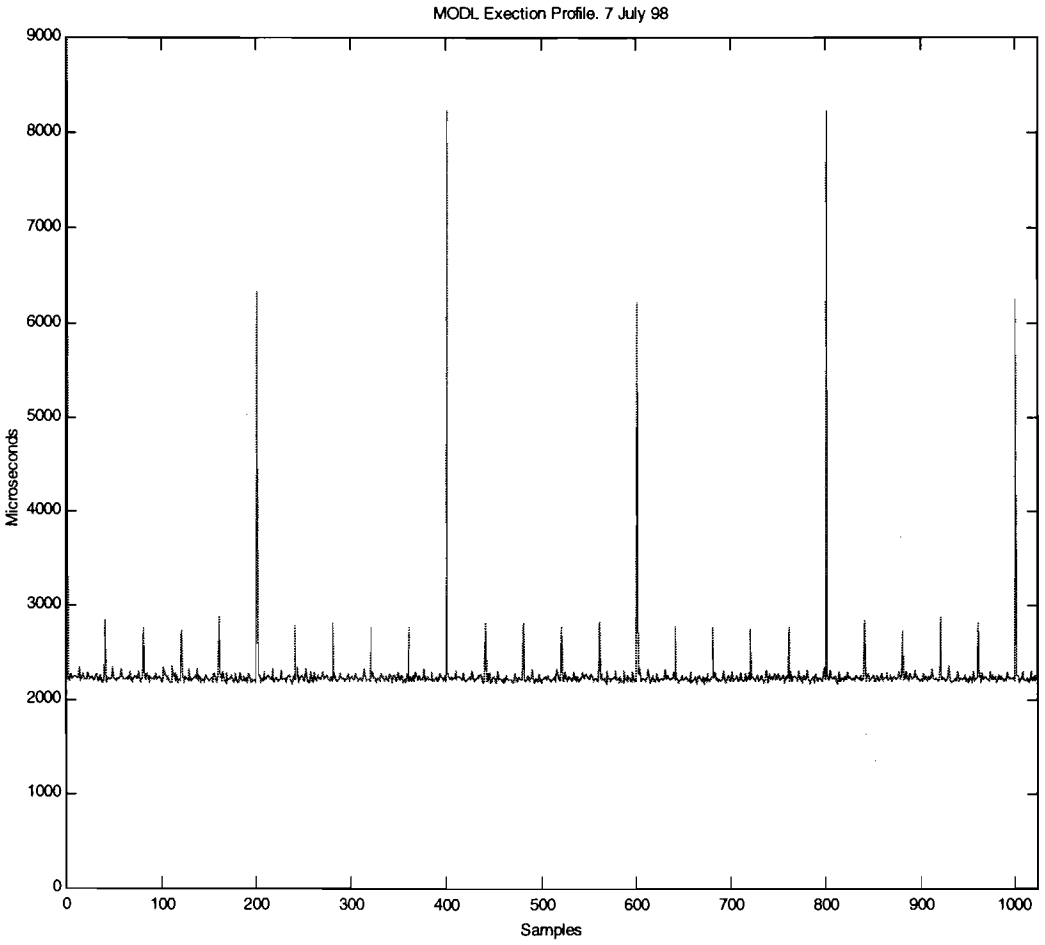
The third thing to mention about the removal of simstruc{} is to explain what the attitude team gained by doing so. Frankly, the newly generated code is more concise (smaller), faster, and easier to understand than the original MathWorks code. Most importantly, the constraining platform compiler's RTX 2010 toolset now compiles the source code. The auto-code developers integrated a preliminary set of the auto-code onto the engineering model of the AIU, and have encouraging results to show. The code performed within timing specifications, taking 22 of the allotted 25 milliseconds to execute, worst case. Since then, the real-time epoch changed from 25 Hz to 10 Hz, relieving concerns over timing performance. There are still space concerns, and the code generator has proven to be not so efficient in that regard. The attitude team believes that the S-Function capability provides an avenue for managing space.

A Single Task with Multiple Rates

As stated above, it is the success of the attitude team's integration of the RTW product on TASTIE that validates our approach. There is an aspect to this success that relates to the removal of simstruc{}. While modifying the code to become simpler and easier to understand, the removal of simstruc{} also disables the generated code from utilizing RTW's multitasking capability. Although a sometimes significant advantage when developing embedded systems, the TIMED attitude team chose not to use multitasking, implementing all auto-generated code as a single task. To be very clear, this does not mean that the auto-generated code handles only one rate. Both the default source code and TIMED team's modifications handle any number of concurrent rates, as specified in Simulink by the analyst.

Removing simstruc{} does negate the use of multitasking, but it simplifies the coordination of the multi-rates within a model. The simplified approach allowed the attitude team to solve a real-time bandwidth problem when first attempting to integrate the auto-code on TASTIE. Figure 6 illustrates an execution profile of the auto-code on TASTIE. The presence of multiple rates is clear with the regularly spaced peaks. The daunting feature in the plot shows the highest peaks, every 200 samples, exceeding the 5 millisecond fundamental task rate. With the default MathWorks files and scripts, the conclusion would be that the RTW auto-

Figure 6:



This is a run-time profile of TASTIE code. The processor load for each rate is visually apparent. The maximum allowable time for any sample is 5 milliseconds, clearly violated by the 1 Hz code. (Recall that the fundamental sample rate is 200 Hz.) This required the separation of the 1 Hz 'task' from the other 'tasks,' easily facilitated with the multi-rate implementation established by RTW, and emulated by the new TIMED implementation sans Simstruc{}.

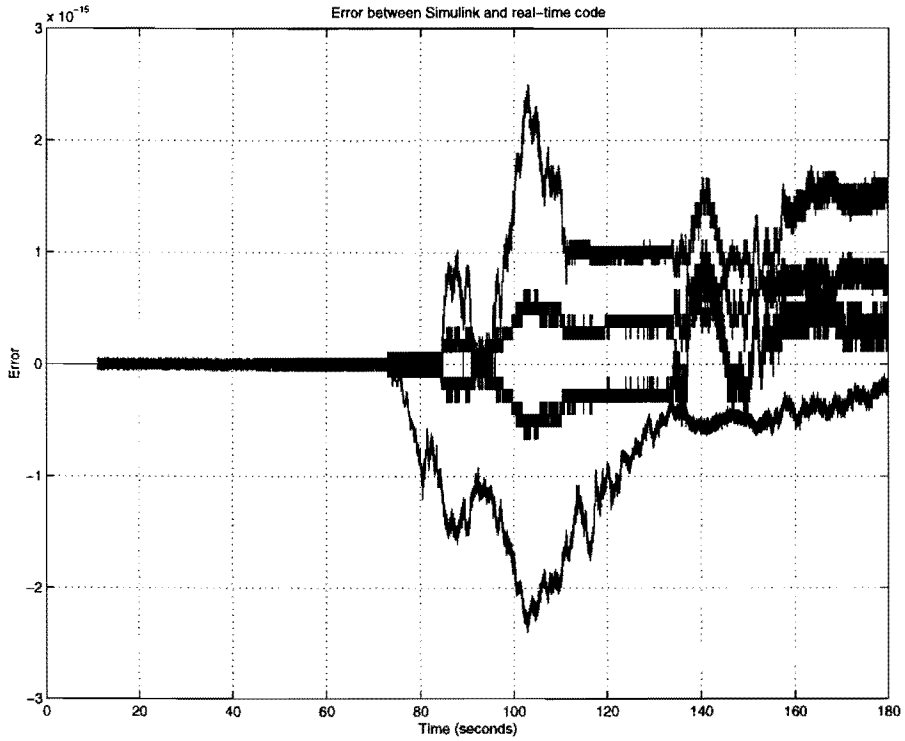
code cannot fit on the TASTIE platform. However, with the new simplified code, the attitude team was able to separate out the 1 Hz auto-code, intact, and place it in another task managed by the OS/9 operating system. This separation of rates, above and beyond the separation of subsystems, is an exception to the general rule that all RTW code is a single task with multiple rates. The AIU code performs within margin as a single task, and the attitude team anticipates no difficulty with performance on the AFC platform.

Testing of Auto-Code Product

At the bottom-most level, all testing on auto-generated code is black box testing. The plan treats each user-defined root-level subsystem as a unit, and conducts separate tests on each. The plan has two phases. The first phase is entirely in Simulink itself, and comprises of stimulating all subsystems, or 'units' completely to follow every branch in the model. This phase produces the data outputs to which all subsequent tests check

their results. Phase two follows a basic philosophy: Equality establishes that if A equals B, and B equals C, then A must also equal C. In this case, A is the Simulink model, whose range is the data outputs of phase one testing. 'B' is the auto-generated code, *as produced by the default TLC scripts, and library source files.* Testers establish the relationship between the internal Simulink model and the generated code, already assumed to be tested by MathWorks' quality assurance. Any differences between the outputs of phase one and this first half of phase two attribute to compiler and machine limitations (assumed benign). Finally, to complete phase two, testers perform, on each respective platform, the same tests on the TIMED TLC script-generated files and source ('C') as performed in Simulink ('A'), and verified on the default auto-code ('B'). Then as A equals C, the internal Simulink model matches the TIMED TLC script-generated code. Together with the phase one testing, this rigorously tests every branch of the flight code.

Figure 7:



This is a plot of the error between the internal Simulink simulation results and the new TIMED generated code results. The plot shows the four quaternion elements over the first 180 seconds of the simulation. At 200 Hz, the plot compares a total of 36,000 samples. Errors mainly come from printf() rounding and resolution in the files that are compared by environment.

Believed to be the most crucial aspect of using auto-generated code, testers match the values of input and output signals on each subsystem (unit) at every fundamental step in Simulink. This means that over the span of the simulation time (currently around 10,000 seconds) each 200 Hz tick-output compares to the last possible decimal place for a given platform. To date, we have seen only machine epsilon variations in output differences. (See Figure 7.)

Program Impact

Using Simulink's RTW solves many problems for APL attitude control analysts. It creates a readily accessible attitude control environment in the same 'language' that analysts use to develop the algorithms. Namely, the graphical interface in Simulink provides a computer language-independent arena for analysts from different backgrounds to converse. An analyst who developed control code for an Ada-based flight code set is now able to converse directly with a C-based developer by removing the analysts from the tedium of flight code pro-

gramming. On the other side, Simulink's RTW unifies the product delivered to flight processor programmers. Using an auto-code generator creates a new position: the auto-code specialist. This person, and it can indeed be just a single person, knows the connectivity between the graphical representation and the auto-code source files. The specialist knows exactly how a "bug" in the generated files traces back to the block diagram. This single person replaces the role of attitude flight code programmer across the entire project. From scratch to working flight code, a single auto-code specialist fills the gap between the analyst and the embedded processor engineers. All of the work presented in this paper summarizes the last half-year of development.

The specialist position has a unique role in the TIMED program. For this demonstration program, the initial investment in the specialist is heavy. From that point forward, however, the specialist only needs to maintain a maintenance role, remaining "on call" if problems arise. This maintenance role continues past the first program into subsequent programs. Future APL missions will utilize the code generator without starting from scratch.

There are however, some serious difficulties with using MathWorks' code generator. First of all, there is no facility in Simulink to 'find' blocks, or to configure them within in the graphical interface. Simulink lacks a vested mechanism for organizing, tracking, and maintaining blocks in a model. All management of Simulink block diagrams is manual. (Of course, after RTW produces the source, traditional configuration management with tools such as RCS and MKS™ resume.) The TIMED attitude control development and subsequent code generation is subject to 'upgrade,' and all RTW enhancements. There is no guarantee that the TIMED TLC scripts are compatible with the next release of RTW. Of course, every program faces such difficulties in the new NASA paradigm of COTS products.

Finally, the most profound impact on the TIMED program has been the effect of using Simulink on the analysts themselves. Using an auto-code generator directly from the analysts' simulations requires a great deal more discipline on the analysts part, perhaps more than they bargained for. Concerns for execution time, code space, signal conditioning, and configuration management essentially all rest on the shoulders of the analysts, who previously relegated those concerns to the flight processor engineer. While helping consolidate their efforts, the auto-code generator introduced a new class of concerns. However, these concerns can only increase the quality and realism of the simulations. Forcing analysts to model spacecraft components such at the 1553 bus and the analog-to-digital converters increases the burden, but may potentially avoid downstream mismatches so common in spacecraft development.

Summary

Analyst discipline in Simulink enables realism in simulation, and a high quality flight code product. The auto-code generator's open architecture empowers the new auto-code specialists to make radical, resource saving changes to attitude flight code while streamlining the process. The TIMED attitude team now develops algorithms in the same environment that produces the flight code. We have essentially eliminated the middle-man. At the current time, auto-code runs on two of the three intended platforms.

There are things we expect to see from the Simulink RTW tools in the future. Namely, the ability to easily re-ingest generated code back into Simulink remains at the top of the list. Furthermore, we anticipate

enhancements to block and subsystem management within Simulink itself. Although specific to APL's attitude team, these desires reflect the growing community of systems designers, not just attitude and control designers, eager to use auto-code generators for test and flight software. The TIMED demonstration of auto-code utility is not a technological step forward as much as it may influence the general acceptance of such tools in the satellite industry.

---

[i] Press, W. H., Numerical Recipes in C : The Art of Scientific Computing Second Edition; Cambridge University Press, New York, 1992; pp.336 - 338