DAVID MCCOMAS

Incorporating Manual and Autonomous Code Generation

Code can be generated manually or using code-generations software tools, buthow do you interpret the two? This article looks at a design methodology that combines object-oriented design with autonom~~ous~~ic code generation for attitude control flight software.

Recent improvements in space flight computers including floating-point hardware, ample EEPROM/RAM, and plenty of CPU power are allowing software engineers to spend more time engineering the applications software. In my case, the application is the attitude control flight software for an astronomical satellite called the Microwave Anisotropy Probe (MAP). The MAP flight system is being designed, developed, and integrated at NASA's Goddard Space Flight Center. The MAP controls engineers are using Integrated Systems Inc.'s MATRIXx for their controls analysis.[1] In addition to providing a graphical analysis environment, MATRIXx includes an autonomo~~us~~ic code generation facility called AutoCode. As the software engineer I was faced with the task of designing the interface between the manually generated flight software and the autonom~~ously~~ically generated C code. This article examines the forces that shaped the final design and describes three highlights of the design process:

- *Defining the manual-to-autonom~~ous~~ic code interface.* The design shields the controls engineers from the flight environment and defines a robust functional interface that has had little change
- *Applying object-oriented design to the manual flight code.* Modeling the control modes using inheritance provides a simple and robust design
- *Implementing the object-oriented design in C.* The implementation of the inheritance hierarchy is not a generic object-oriented implementation in C, but it proved to be adequate for MAP's requirements

Attitude determination uses sensor measurements to update the onboard estimated attitude which is supplied to the controller subsystem.

MAP attitude control

Figure 1 shows a simplified high-level block diagram of MAP's flight control software. Sensors measure spacecraft position and rates. Attitude determination uses sensor measurements to update the onboard estimated attitude which is supplied to the controller subsystem. The desired spacecraft attitude is either supplied by mode management or internally computed by command generation. Attitude error computes control errors for the control law, based on a combination of sensor measurements, estimated attitude, and commanded attitude. The control law computes control torques which are output to the actuators. The shaded portion identifies the controller subsystem which has been designated for autocode. (The remainder of the article will use autocode to refer to both the tool and the autonom~~ously~~ ically generated code.)

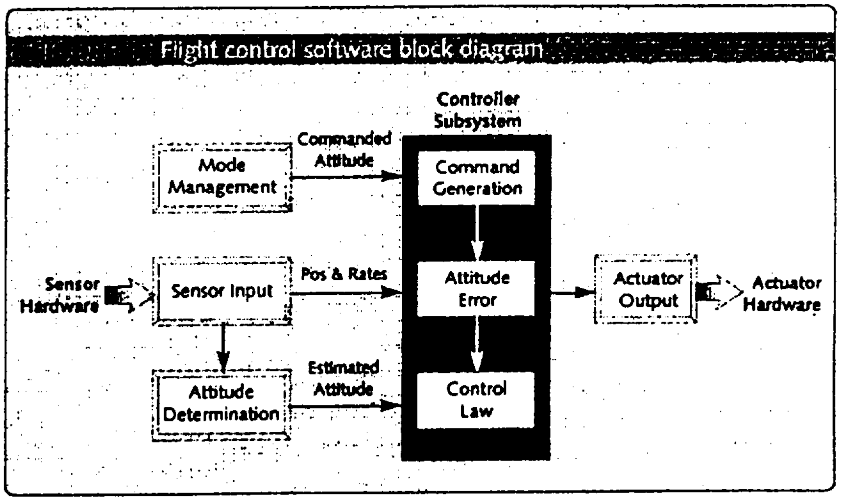
To understand the object-oriented design within the context of this article, two other features of MAP must be described: the sensors and actuators, and the operational modes. MAP uses the following sensors and actuators for attitude determination and control:

- Inertial reference units (IRU)—Measure angular changes in MAP's position. Spacecraft body rates are derived from the incremental angular measurements
- Digital sun sensor (DSS)—Provides accurate measurements (<0.01 degrees) of the sun's position within a 64 degree square field of view
- Coarse sun sensors (CSS)—Provide coarse measurements (<10 degrees) of the sun's position. The CSSes are mounted to provide 4 p steradian coverage
- Star tracker (ST)—Provides an estimated attitude derived from star measurements
- Propulsion control system (PCS)—Provides external torque to the spacecraft via hydrazine-fueled thrusters
- Reaction wheel assembly (RWA)—Provides spacecraft momentum control via three reaction wheels

MAP uses five operational modes to achieve its mission goals. Modes are defined in terms of operational objectives, spacecraft control objectives, and performance criteria. Each mode specifies a set of sensors and actuators and a control subsystem configuration. MAP defines the following modes:

- Sun Acquisition (SA)—Uses IRUs, CSSes, and the RWA to acquire a sun-pointing, power, and thermally safe attitude within 20 minutes from any initial attitude
- Inertial (IN)—Uses IRUs, DSS, ST, and the RWA to acquire and hold a fixed commanded attitude
- Observing (OB)—Uses IRUs, DSS, ST, and the RWA to perform a scanning pattern. Observing is the only mode used for collecting science data
- Delta-V (DV)—Uses IRUs and the PCS to perform spacecraft maneuvers. Delta-V is used for trajectory management to get to the Sun-Earth L2 point approximately 1.5 million km from the Earth (away from the sun) and for L2 stationkeeping
- Delta-H (DH)—Uses IRUs and the PCS to perform momentum unloading

Flight control software block diagram



Scope the interface

Business, as well as technical forces, shaped the boundary of the autocode subsystem. On previous missions, a high fidelity simulation (HIFI) written in Fortran was used to develop the control algorithms which were documented via a hand-written algorithm document. Autocode changes this paradigm by forcing the HIFI design to assimilate enough of the FSW environment to allow the autocode to be used directly by the FSW. Autocode's scope dictates how much of the flight environment needs to be modeled by HIFI.

MAP's FSW tasking architecture is built on an existing "software bus" which placed limits on autocode's scope. The software bus provides standardized packet-based intertask communication and insulates applications from the real-time operating system. This heritage immediately limited autocode to an intra-task scope and ISI's RTOS, pSOSystem, was not even considered. The flight controller pictured in Figure 1 is suitable for a single task because all of its components execute at 1Hz and have fairly strong data cohesion.

Additional design decisions further narrowed the autocode scope to the shaded controller subsystem. It has a small and simple set of inputs and outputs. All I/O can be performed in the spacecraft body frame. The controller subsystem does not directly interface to the following FSW subsystems: sensor/actuator hardware, ground command inputs, error message output, and fault detection. Therefore, the HIFI does not need to emulate these FSW facilities. Attitude determination shares many of the same attributes as the controller subsystem with respect to being suitable for autocode, but it was not chosen for autocode since MAP could adapt an existing attitude determination subsystem from a previous mission.

TABLE 1 Contents of input and output

Input
[illegible]
Wheel Speeds
Estimated Attitude
[illegible]
Measured Sun Angle
Mode Control Flags
Output
[illegible]
PCS Command Counts

Limiting the scope of autocode to the controller subsystem coincides with Goddard's business philosophy as well. MAP is Goddard's first mission to use an autonomous code generator for its FSW, and prior to MAP, Goddard has had minimal experience with ISI's code generator. Coupled with MAP's short development schedule, this small, well-contained subset of the FSW is a good way to minimize the risk factor. In addition, the controller subsystem has a high algorithm-to-logic ratio which maximizes autocode's benefits of speeding up the process of documenting and coding the algorithms from HIFI and in reducing errors during the translation process. However, MAP's return on investment is limited due to the learning curve and the small scope of the autocode relative to the size of the entire FSW effort.

Autonomous code environment

MAP is using autocode's basic features since we are generating discrete procedures without multitasking. The output of autocode is simply a collection of C functions. Using autocode's template language, I have customized autocode to output each function into a separate source file with a corresponding header file. This helps isolate and identify exactly what code has changed as a result of an algorithm change. This strategy may be useful after our scheduled development period when we need to access the testing effort required to verify a late algorithm change.

The collection of C functions can be conceptualized as a single object. Figure 2 shows a class diagram representation of the autocode interface that must be managed by the manually coded FSW. Only two autocode functions need to be called. `Initialize` needs to be called once during system initialization. `Execute` is called each control cycle and it manages calling the subordinate autocode functions. The `Input` structure is loaded prior to calling `Execute` and the `Output` structure contains the results. The contents of `Input` and `Output` are as shown in Table 1.

Object-oriented design

As I mentioned, control modes specify a set of sensors and actuators to be used and the control subsystem configuration. Inheritance works particularly well to abstract common attributes and behavior shared among the control modes. Figure 3 shows the inheritance diagram used as a model for the manually coded portion of the controller. The modes are first classified according to what actuator is used for control and each actuator controller is subdivided into specific controllers.

The base controller class provides three functions: `New`, `Delete`, and `MonitorPerformance`. The italicized functions are virtual functions and descendants provide the implementation. `New` and `Delete` are used to instantiate and destroy controllers, respectively. `MonitorPerformance` is

FIGURE 2 Autocode class diagram

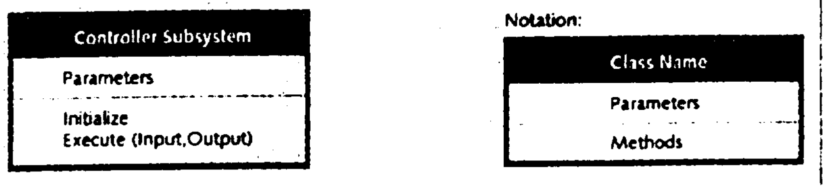
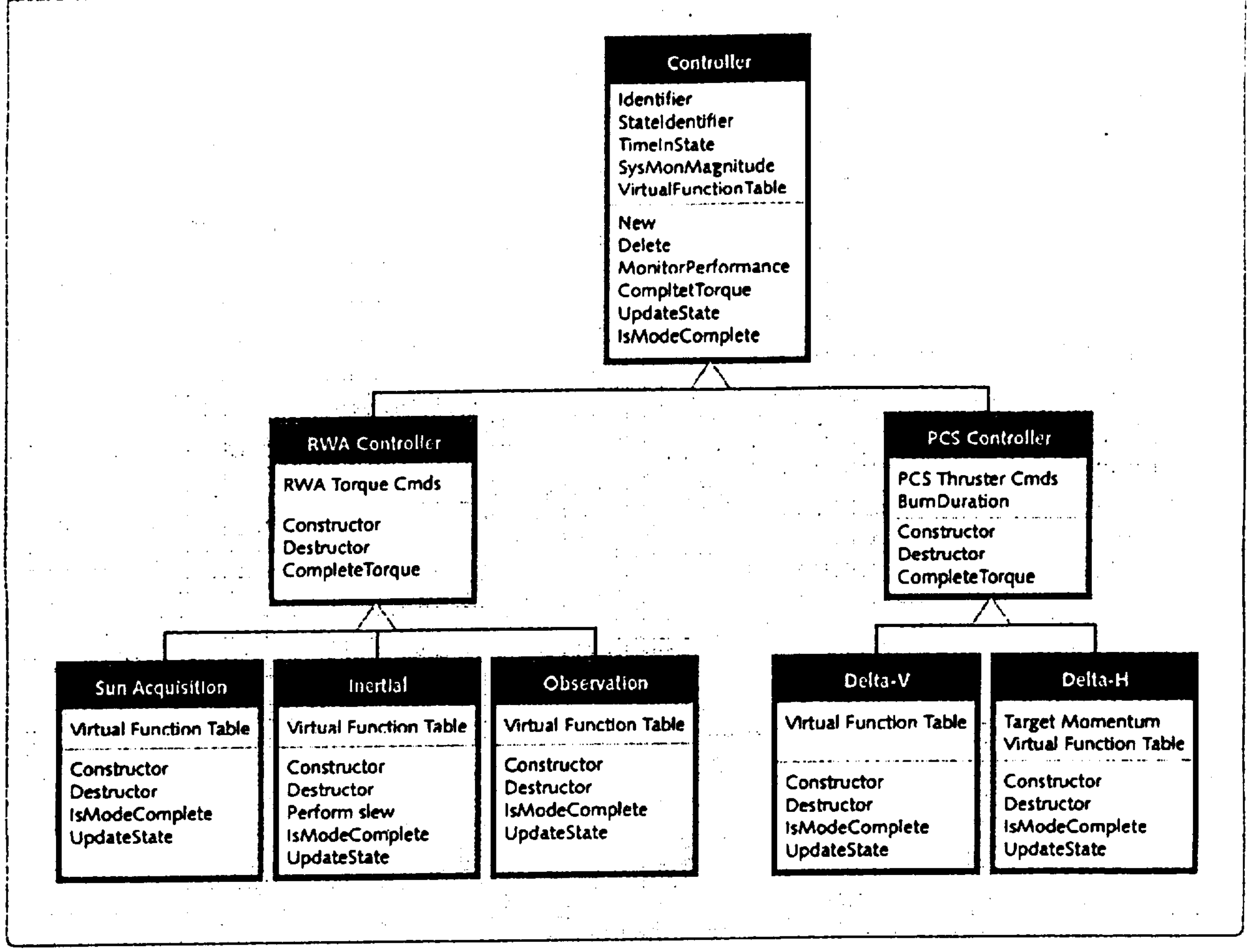


FIGURE 3 Controller inheritance diagram



similar to a protected function in C++ and is used by descendant classes to monitor body rates, body rate errors, and attitude errors. Notice that ComputeTorque is implemented by the RWA and PCS controllers and not the five control modes. ComputeTorque calls autocode's Execute function, passing mode-specific control flags that are defined when a mode is constructed. Finally, IsModeComplete and UpdateState are implemented by each controller because they address state information that is unique to each controller.

Since the base Controller class defines a common interface for all of the controllers, the code that manages the controllers is identical regardless of which controller is executing. This proved very useful during unit testing since the test driver is identical to the FSW that manages the controllers. The Input and Output data structures used by the autocode are managed by both the Controller class and the RWA and PCS controller classes. Although management of the autocode interface is not encapsulated within a single class, it has proven to be a robust implementation, resilient to ripple effects due to changes in the autocode interface. Inheritance has also kept the functions relatively small and simple so they have been easy to understand and test.

Implementation

I did not take a general approach toward implementing object-oriented concepts in C. Virtual function tables were manually created and no dynamic memory allocation is used. Listing 1 shows the essential data type definitions for the abstract controller class. The ATTCTL_RWA and ATTCTL_PCS structures are used by the RWA and PCS controllers and access control is managed by the programmer. In C++ these data structures would be defined as part of the RWA and PCS classes and the compiler would enforce data access control. The New function loads the ATTCTL_VTBL structure. A pointer to a controller's virtual function table is supplied as a parameter to New. This implementation relies on the fact that multiple controllers cannot exist simultaneously.

Virtual function table

```
/* Virtual Function Table */
typedef void  (*ATTCTL_CONSTRUCTOR)   (void *);
typedef void  (*ATTCTL_COMPUTE_TORQUE)  (void);
typedef void  (*ATTCTL_DESTRUCTOR)    (void);
typedef Boolean (*ATTCTL_IS_MODE_COMPLETE) (void);
typedef void  (*ATTCTL_UPDATE_STATE)   (void);
typedef struct {
  ATTCTL_CONSTRUCTOR     Constructor;
  ATTCTL_COMPUTE_TORQUE  ComputeTorque;
  ATTCTL_DESTRUCTOR     Destructor;
  ATTCTL_IS_MODE_COMPLETE IsModeComplete;
  ATTCTL_UPDATE_STATE   UpdateState;
} ATTCTL_VTBL;

/* RWA mode structure: Only meaningful during RWA modes */
typedef struct {
  float Torque[NUM_WHEELS];
} ATTCTL_RWA;
/* PCS mode structure: Only meaningful during PCS modes */
typedef struct {
  Unsigned16 Counts[NUM_THRUSTERS];
} ATTCTL_PCS;
/**** Abstract controller class ****/
typedef struct {
  . . .
  ATTCTL_RWA  Rwa;
  ATTCTL_PCS  Pcs;
  ATTCTL_VTBL Function;
} ATTCTL_CLASS;
```

ComputeTorque's implementation

```
void AttCtl_ComputeTorque(void) {
  // Load INPUT structure with Measured BodyRate, Wheel Speed, Sun Angle
  // Load INPUT structure with estimated and commended attitude
  // Call AutoCode Execute(INPUT,OUTPUT)
  (*AttCtl.Function.ComputeTorque)(); // Call RWA/PCS virtual ComputeTorque
function
  // Load AttCtl.SysMomMag using OUTPUT data
} /* End AttCtl_ComputeTorque() */
void AttCtl_RwaComputeTorque(void) {
  // Load AttCtl.Rwa structure using AutoCode's OUTPUT
} /* End AttCtl_RwaComputeTorque() */
```

Listing 2 shows ComputeTorque's implementation. AttCtl_ComputeTorque loads the dynamic portions of the autocode Input data. The portions of the Input structure that are static during a controller's lifetime are loaded when a controller is constructed. AttCtl_ComputeTorque invokes the autocode followed by a call to the RWA or PCS ComputeTorque virtual function. The RWA or PCS function processes the mode-dependent portion of Output as shown in AttCtl_RwaComputeTorque. Finally, the mode independent portion of Output is processed by AttCtl's ComputeTorque.

Some downsides exist to using the autocode in this fashion. The Input structure to autocode includes mode control flags which summarize information that is already captured by the class inheritance structure. Autocode contains logic to execute different forms of the controller based on the Input mode control flags. If the entire system were designed as a whole, the conditional blocks of the autocode controller would exist as functions within the individual controller classes.

Another drawback to AutoCode is that it can be inefficient with respect to both memory and speed. Autocode works on each graphical element as it produces code. Many blocks are translated into functions with large parameter lists. Functions are good for traceability from code to design but may be bad for code optimization. There is an inline procedural block feature that allows blocks to be generated inline with the current block's code. This helps, but lumping code into a single function is not always desirable. For MAP, we are generating each "superblock" as a separate function contained in a separate file. This allows algorithmic changes to be configuration managed at the source file level. The additional function overhead is acceptable since we aren't close to our CPU or memory budgets.

MAP is also taking a conservative approach with respect to testing the autocode. To maximize savings in test time, autocode would be treated as a black box during FSW unit testing. The analyst would test the algorithms in HIFI and the autocode would be passed to the build/acceptance test team after being integrated with the rest of the FSW. MAP has altered this

Further MAP reading

MAP's mission is to probe conditions in the early universe by measuring the properties of the cosmic microwave background radiation over the full sky. These measurements will help determine the values of Big Bang cosmological parameters and determine how and when galactic structures formed. MAP will maintain a halo orbit about the Sun-Earth Langrange point 1.5 million kilometers from Earth (away from the sun). The attitude control system will maintain a .464 rpm spin rate about the *z*-axis during observations. For more information see the December 1997 issue of *Discover* magazine or visit the MAP Web site at *inap.gsfc.nasa.gov*.

direct approach. For early builds, autocode is being unit tested as a black box on a PC using test data captured from HIFI without full path (white box) testing. For the final build, full path unit testing on the flight hardware will be performed. This approach allows us to take advantage of the quick design-to-test time for early builds when algorithmic changes are most likely to happen. However, it is based on the assumption that the full path unit testing will not uncover substantial coding problems late in the project schedule.

Lowering the barrier

Autocode has forced us to formally define our controller subsystem interface because it's a separate physical and logical entity. We may not be exploiting all of the features of autocode but limiting the scope has served us well in terms of meeting schedule and mitigating risks. Autocode has affected our traditional way of doing business. On previous missions, the software engineers have worked fairly independently of the controls analysts, using an algorithms document as a formal communications mechanism. Autocode has successfully lowered the cultural barrier, bringing the software engineers into the controls analysis environment, and the controls engineers have become more cognizant of the flight software environment. This symbiotic relationship helps scheduling since the dependency flow is no longer unidirectional from the analysts to the software developers.

I consider this project to be transitional with regard to how future projects may be developed. Advances in flight software architectures, coupled with advancements in onboard computing resources, should allow for object-based software construction. Standardized object communications mechanism such as CORBA may be a viable option for flight controllers as they mature. The Distributed Object Computing group at Washington University is a good example of the work being done in this area.[2]

Once a business establishes an object communications layer, reusable object libraries can be developed. Reusable objects may be archived in a format that is usable with respect to a developer. For example, a reusable controller written in C does not have much value to analysts that work in a graphical environment. The analysts would prefer to have a reusable controller in a form that can be easily manipulated within their development environment. Autonomous code generation tools would allow reusable libraries to exist in different formats, as long as the tools produce objects that conform to an object communications standard. Regardless of what the future may hold, the design of high quality interfaces will be a critical factor in successfully developing and using object libraries. MAP's design is first step towards achieving this goal.

David McComa's bio will go right here.

References

1. Integrated Systems, www.isi.com

2. *siesta.cs.wustl.edu/~schmidt/research.html*

RESOURCES

Executable code related to this article is available at *www.embedded.com*.