Design Patterns for Avionics Control Systems

Doug Lea
SUNY Oswego & NY CASE Center
DSSA Adage Project ADAGE-OSW-94-01

DRAFT Version 0.9.6; November 20, 1994

0 Introduction

A design pattern[1, 5, 8] is an encapsulated set of solutions, alternatives, rules that lead to solutions, and/or process guidelines for dealing with a design problem arising in a particular context. Each design pattern relies on, results in, and/or interacts with other contexts, problems, and solutions addressed in other patterns.

The patterns in this document combine observations, reinterpretations, rational reconstructions, and redesigns of Avionics Control Systems within the realm of the DSSA ADAGE project[3]. An Avionics Control System (ACS) is the main navigation system of an aircraft. An ACS continuously collects sensor data to estimate actual state of an aircraft, computes desired aircraft state with respect to guidance modes, and performs actions that advise pilots and/or directly manipulate aircraft effectors in ways that bring actual and desired state in closer agreement.

These patterns describe domain-specific architecture concerns and steps in the construction of an ACS using a minimal vocabulary (e.g., ‘‘components’’, ‘‘interfaces’’, ‘‘functions’’, ‘‘attributes’’), and with minimal commitment to how these should be expressed within any particular design method, notation, engineering tool, or development process. However, because of the critical impact of Avionics Control Systems on human safety, it is essential to capture the resulting designs into appropriate formalisms and semi-formalisms that can be analyzed and reasoned about.

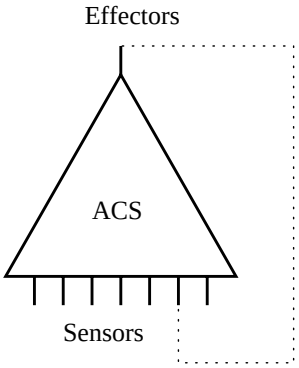
Most patterns address only the architectural forms of components, interfaces, connections, and protocols, delving into details only when they impact overall design. These patterns may be used to generate instances of a family of concrete architectures and designs that may be implemented using a number of different languages, tools, and systems. Because of the diversity of platforms and programming languages used in avionics systems, implementation details are notably absent. This collection of patterns also omits sufficient description of signal processing and control algorithms, device characteristics of radar systems and other navigation hardware, pilot instrument and user interface design, and mechanical effector systems necessary to actually construct an ACS from scratch. Information and guidance on such matters must be obtained from other sources (e.g., [6, 13, 12]). Given the extensive history of avionics system design, the most likely users of this set of design patterns include people learning about the design space leading to different avionics architectures, developers redesigning existing systems, and those building new families of components.

Most entries take a simple form. They start with a description of the context, problem-space, and constraints, and then describe associated **Design Steps**, normally in a prescriptive manner, and sometimes subdivided into sets of related steps and concerns. Each pattern references others through hypertext links (or cross-references in the paper version of this document).

1 Structuring Avionics Control Systems

The top-level functional requirements and structural constraints on aircraft control systems can be categorized into two types of concerns: *Flow* and *Models*:

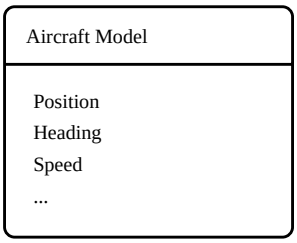
Flow. An ACS continuously collects a large number of inputs, including positional and environmental data from external sensors, state data about aircraft components and controls, and desired flight objectives and other control information. It ‘‘converts’’ these into a small number of outputs, primarily displays alerting Pilots about suggested control actions, and direct manipulations of aircraft effectors.



Nearly all system-wide top-level information flow in an ACS is inherently *upward*, from sensors to displays and effectors. Overall flow resembles that maintained, for example, in neural network designs. Even most ‘‘feedback’’ flow falls under this pattern. For example the state of a wing is obtained from sensors, not just assumed to be what its last effector command requested.

The functional form of flow in an ACS is *transformational*. Information continually changes form across paths from input to output. To be useful, nearly every representation across the overall flow of an ACS must be transformed according to some kind of filtering or conversion algorithm. For example, sensor data must be transformed from often-messy device-dependent readings into estimates of where the aircraft is. *Filter* functions represent the ‘‘gates’’ that connect stable representations.

Models. To carry out its functionality, an ACS must maintain coherent, accurate models of the world.



From a structural perspective, an ACS may be seen as an instance of a Model-View-Controller (MVC) design[7]. An ACS maintains a ‘‘reactive’’ model of the world that is controlled via updates in response to new data sources, and is viewed ultimately by displays and manipulators. MVC frameworks (along with numerous minor variants) split the roles of model-maintenance, input ‘‘sources’’, and output ‘‘sinks’’ across different components in order to isolate variation in how each of the associated responsibilities may be accomplished. In an ACS, there is scarcely even an alternative to this structure. Modeled aircraft properties may rely upon any or all of a number of different sensors, controls, estimation rules, guidance

modes, and/or policies. These properties may be used to generate outputs in any of several ways. It would be inconceivable to construct a set of components that merge all possible combinations. Instead, components of each type are linked together to serve particular purposes.

Because of the transformational nature of ACS systems, the linkage between models and other components is almost never a simple matter of arranging connectivity to propagate value updates. Instead updates must be performed via invocation of suitable filtering functions. These transformations may take any number of functional forms. It is even possible to transform sensor data in multiple ways to obtain multiple estimates. This leads to a second kind of separation principle in ACS design: Data transformation functions must be independent of the components that use them in the course of updating representations.

Algorithmic concerns also govern the top-level categorization of different kinds of models. The ''topmost'' model in an ACS is an *Action* model, representing desired effector and display settings. Essentially all known algorithms for computing these representations employ variants of a standard strategy. Estimated properties of the actual world are compared with ideal or desired properties that may be further transformed into changes in effector and/or display settings that reduce these differences. Thus, *Error* models representing differences between actual and desired state play a central role in ACS systems. Because error models are used to effect particular changes in the aircraft, they must take a very simple form, typically representing only a few summary attributes. To support the comparator algorithm, other ACS models are subdivided into two similar yet mostly independent categories, *Navigation* models, that represent the actual state of the world, and *Objective* models, that represent desired state.

Design Steps

1. Construct components maintaining Models (§3), each Representing (§2) the state of the aircraft and significant aspects of its environment:

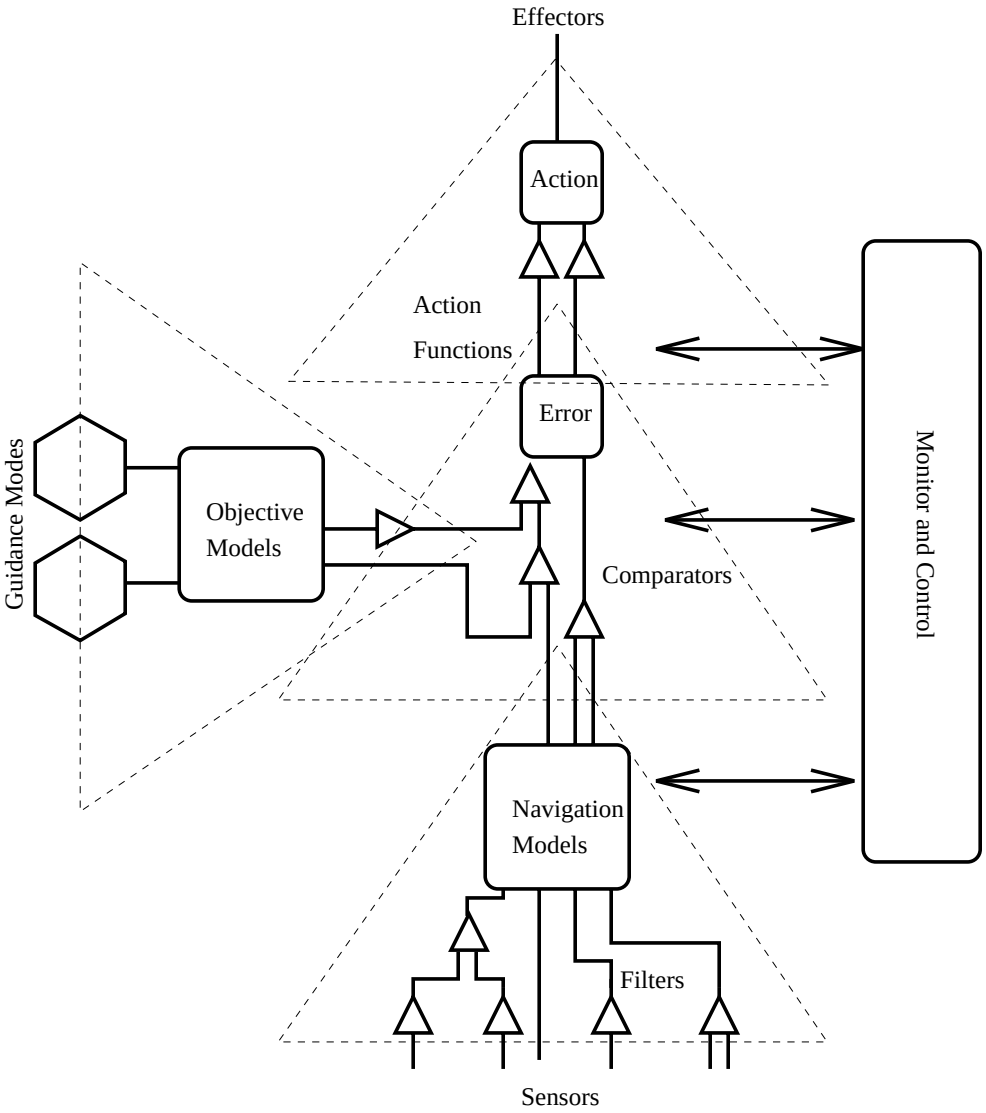
    - Navigation Models (§4) representing estimates about the *actual* state of the aircraft and other real-world objects, continuously updated from sensor data.
    - Objective Models (§5) representing *desired* states of affairs, that may also change over time.
    - Error Models (§6) representing only those differences between actual and desired state useful for directing flight.
    - Action Models (§7) representing desired effector and display settings in a form suitable for manipulating aircraft hardware.
    - Staged (§8) helper components serving as intermediate models, interposed when necessary or convenient for estimation purposes.

2. Construct components serving as Sources and Sinks (§9) for models:

    - Sensors (§10) that interact with the external world to provide inputs.
    - Effectors (§11) that transduce representations of desired settings into mechanical actions.

3. Construct computational signal-processing components:

    - General and special purpose Filters and Transforms (§12) that transform data on its way from place to place, where each function is designed independently of its use in any particular estimation operation across model components.



- Guidance Mode (§13) components that compute objective model values according to particular rules.

4. Construct policies, algorithms, and components that tie components together:

   - Flow (§14) policies that govern the mechanics of information flow through the system, along with Monitoring and Control (§15) facilities to dynamically track, enable, and alter constituent components.
   - Corresponding Update (§16) protocols among components along flow paths.
   - Configuration (§17) strategies and tools for putting together an instance of the system. These may result in some of the above components being merged, or even ‘‘optimized away’’, all but disappearing at the implementation level.

2 Representing State

The heart of an ACS is a set of Models (§3) of the world; pulsating, active models with a purpose. Of course, an ACS need only model those properties of that small subset of the world that are useful for Effecting Flight (§11), and/or are knowable from Sensors (§10). Thus, the representational techniques used in an ACS are products of top-down and bottom-up forces:

- The top-down forces stem from ultimately the information needed to advise pilots and control Effectors (§11)
- The bottom-up forces stem from the need to Update (§16) models using values derived ultimately from Sensors (§10) and other data source inputs.

However, there is not much need for reconciliation. All of these components intrinsically rely upon values represented on scales that are ultimately meaningful in terms of physical laws about bodies, motion, and change. These abstractions (relative position, velocity, etc.), usually termed *state vectors*, are required in order to arrive at guidance decisions, are the quantities that devices try to measure, and thus must serve as the basis for designing the representational, or *attribute* structure of models.

A state vector is a collection of physically meaningful values, representing measurements or settings of some sort; for example: Pitch, Roll, Position, Velocity, Acceleration, Attitude, Altitude, True Heading, Mach number, True Mean Sea Level Altitude, Indicated Airspeed, True Airspeed, Ground Track Angle, Magnetic Heading, Ground Speed, Angle Of Attack Bank, Angle Drift Angle.

The main reason that state vectors are used in guidance systems is that the abstractions themselves are so tractable mathematically. For example, so long as units of measurement are consistent, you can add two three-dimensional points together whether these represent Aircraft positions or Snail positions.

ACS model attributes are represented by state vectors, including vectors of vectors, and so on. State vector types are not restricted to descriptions of objects as they are in the world, but also include information about the future and past. For example, from position, heading, speed, and so on, they implicitly represent where an aircraft will be in the next instant. They may also represent historical values to help determine trends over time.

The second collection of design constraints on representational techniques rests on the kinds of algorithms used in an ACS. Most signal processing functions and their computational implementations do not differ in form across different units of measurement of their arguments and results. For example, a matrix of real numbers may be inverted in the same ways whether it represents a series of magnetic heading readings or a set of correlations. Nearly all Filtering and Transformation (§12) signal-processing algorithms are of this form. They accept structured collections of numbers, all represented using an arbitrary but common unit of measurement, and return others.

Finally, because most ACS models represent *estimates* of the state of the world, there are bottom-up, local, and top-down forces influencing the calculation and use of these representations with respect to their intrinsic *accuracy*, the computational limitations of estimation techniques, and/or system-wide control of the ACS.

An ACS must possess a lot of redundancy to achieve fault-tolerance. This redundancy cannot always be implemented simply by replicating components. For example, two identical radar-based sensors would

tend to be equally adversely affected by extreme weather conditions, thus defeating much of the purpose of redundant support. Instead, most redundancy is accomplished through the use of multiple kinds of sensors, estimators, models, modes, and effectors. However, not all components are equally reliable or accurate. For example, Sensors (§10), and/or communications channel connecting them can break. Even when not broken, they can occasionally report completely wrong values. Similarly, different Guidance Modes (§13) provide more accurate estimates of desired state than others. The act of Updating (§16) model attributes can introduce additional uncertainties. Different Filtering (§12) algorithms provide better model attribute estimates than others. Sometimes, a value cannot be computed at all due to computational limitations of filters that require input data to be of a certain minimum quality in order to produce transforms. Additionally, pilots and/or Supervisory (§15) components may disable (or ''disarm'') certain components or subsystems, for any reason.

Design Steps

Use multiple levels of representation of state in an ACS, varying in how values are ''tagged''. Different forms of representation serve as the basis for defining attributes of different kinds of Models (§3), and are the data types used in Filter (§12) functions and Update (§16) operations:

- Values assigned a Measurement Type (§2.1) and associated physical meaning.
- Measurements and estimates tagged with indications of Accuracy (§2.2) and use.
- Collections of Dimensionless Values (§2.3) used in signal-processing algorithms.

2.1 Units and Types

Determine a set of measurement types via analysis of the physical properties that are represented and manipulated in the ACS [3]. Each type should be interpretable in terms of physically meaningful units:

1. Create, name and organize a collection of basic value types in terms of their abstract physical meaning, *not* the entities that they represent. Assign generic, standardized names to types and fields.
2. Define these types from the bottom. The primary dimensions for aviation quantities are Mass, Length, Time, and Temperature. Other derived types are composed as expressions on these primary units along with dimensionless values such as Angle. For example, velocity has the dimension Length / Time, and XYZ position is a 3-element vector of Length. Layer composite state vector types (e.g., *position*) as records and/or other composite types containing elements of these base types.
3. Minimize and standardize the number of units of measurement. When possible, associate a single unit of measurement (e.g., cm, sec, radians) to each type. Assign consistent implementation types (integer, float, etc) to each unit.

2.2 Accuracy and Arming

For each representation subject to uncertainty or disabling, maintain associated ''qualifiers'' or subattributes:

Merit. A figure describing its degree of accuracy, confidence, quality and/or reliability. Ideally, all merit figures should be encoded in a common Unit of Measurement (§2.1).

Armed. An on/off indication of whether it is enabled; that is, whether it can/should be used in other computations or be propagated to other components.

There are several variants and simplifications. For example, in some cases, these two qualifiers may be folded together as a single value. Also, merit and/or armedness values may sometimes apply to all attributes maintained by a given model component. And some values may always be known with the same degree of accuracy, in which case these figures may be statically or even implicitly represented.

2.3 Dimensionless Views

Creating a *dimensionless view* of one or more representations allows you to ignore units, attribute names, and model component types when applying generic signal processing algorithms, while still representing them for the sake of maintaining models.

For each kind of view, define a type. There are typically a relatively small number of these types, with very familiar names, for example, *Pair, Vector* and *Matrix*, specializations such as *DiagonalMatrix*, along with associated operations such as *add*, *invert*, etc.

There are two strategies for obtaining and using dimensionless views from model attributes, direct, and indirect. They differ in the same way that ''value-based'' and ''reference-based'' representational conventions differ in programming languages. Both strategies support minor simplifications in those cases where views are always ''readonly''; that is when values are used for computations but never fed back in a form suitable for updating their sources:

Direct. For each attribute of a model or collection of models, define a *projection* function that packs the desired value of the desired attribute into a view. These are special kinds of conversionless Converters (§12). Unless the views will always be ''readonly'', define the corresponding inverse *pack* operation that converts back into original form.

Indirect. Define each entry of the view to be some sort of reference to a particular attribute of a particular kind of representation object. Each value access must invoke the corresponding accessor, and unless they are readonly views, each modification must invoke the appropriate update operation.

Filter (§12) functions based on direct versions are simpler to compose, while those based on indirect versions are simpler to use in Model Updates (§16). So neither is always best -- choices rest on Filter (§12) design, component Connections (§16.2) and Configuration (§17) matters.

3 Decomposing ACS Models

The role of an ACS model component is to:

- Maintain a consistent representation of an aspect of the world.
- Update this representation in response to new information from ‘‘source’’ components.
- Alert other ‘‘consumer’’ components about changes.
- Alter behavior in response to system-wide control components.

Many sets of physically meaningful Values (§2) (e.g., heading versus position) used in an ACS are nearly independent of each other, both physically and computationally. There are several reasons that corresponding models of ‘‘macro’’ objects such as the Aircraft itself need not, and should not be constructed as single components: Most updates and queries are localized to particular, known, independent aspects of state. The associated mechanics are simpler to arrange for small, independent submodels than a monolithic component maintaining all possible attributes. Many attributes must be represented in multiple ways. Different ACS systems radically differ with respect to parts, so there is no single, generic compositional model. New attributes reflecting new sensor capabilities, guidance modes, or effectors may need to be added across successive versions of the system. And small, localized components are simpler to Configure (§17) in accord with the special physical, resource, and performance demands of any particular instance of an ACS.

Design Steps

Decentralize models. While they may possess aggregate conceptual interfaces[9], each kind of ACS model should be implemented as a loosely connected *bundle* of submodels, each responsible for maintaining a small independent set of attributes representing related, physically meaningful abstractions.

The resulting collection of attributes and operations for each kind of model or submodel describes a set of interfaces that may be implemented in many ways. Many kinds of models will share many aspects of common interfaces, which can be factored via interface inheritance[4] to arrive at more compact and useful designs.

However, the following steps do not themselves generate interfaces, only sets of attributes, connections, operations, and protocols that must be arranged into common interfaces according to roles and usage. In fact, because ACS model interfaces are uncomfortably sensitive to policies surrounding control Flow (§14), Monitoring (§15), and Configuration (§17), it is a good idea to design attribute structure early, but to return to operation interfaces only after considering these issues.

1. Represent attributes in terms of State Vectors using common Measurement (§2.1) units.
2. Ensure that attributes represent information that is estimable from source components and useful to consumer components. Try to maintain the same (or a common subset of) attributes, in the same way, across different stages.
3. When attributes must be multiply represented or are otherwise interdependent, either within or across models, ensure Consistency (§16.3) among them.

4. Maintain static knowledge (e.g., constraints on possible attribute values, defaults, fixed properties of the entity being modeled, physical constants and laws) in a form usable in the course of updates and access. Different components should represent the same static knowledge in the same ways. One way to ensure this is to pool static knowledge into common database-style components with access operations that are invoked in a uniform manner across various models.

5. Maintain and evaluate the degree of Accuracy (§2.2) surrounding each attribute that is subject to uncertainties in estimation.

6. Maintain an indication of whether the information represented by an attribute is Enabled (§2.2) for use by other components, as Controlled and Monitored (§15) by system-wide components.

7. Maintain those Connections (§16.2) needed to communicate with sources and consumers.

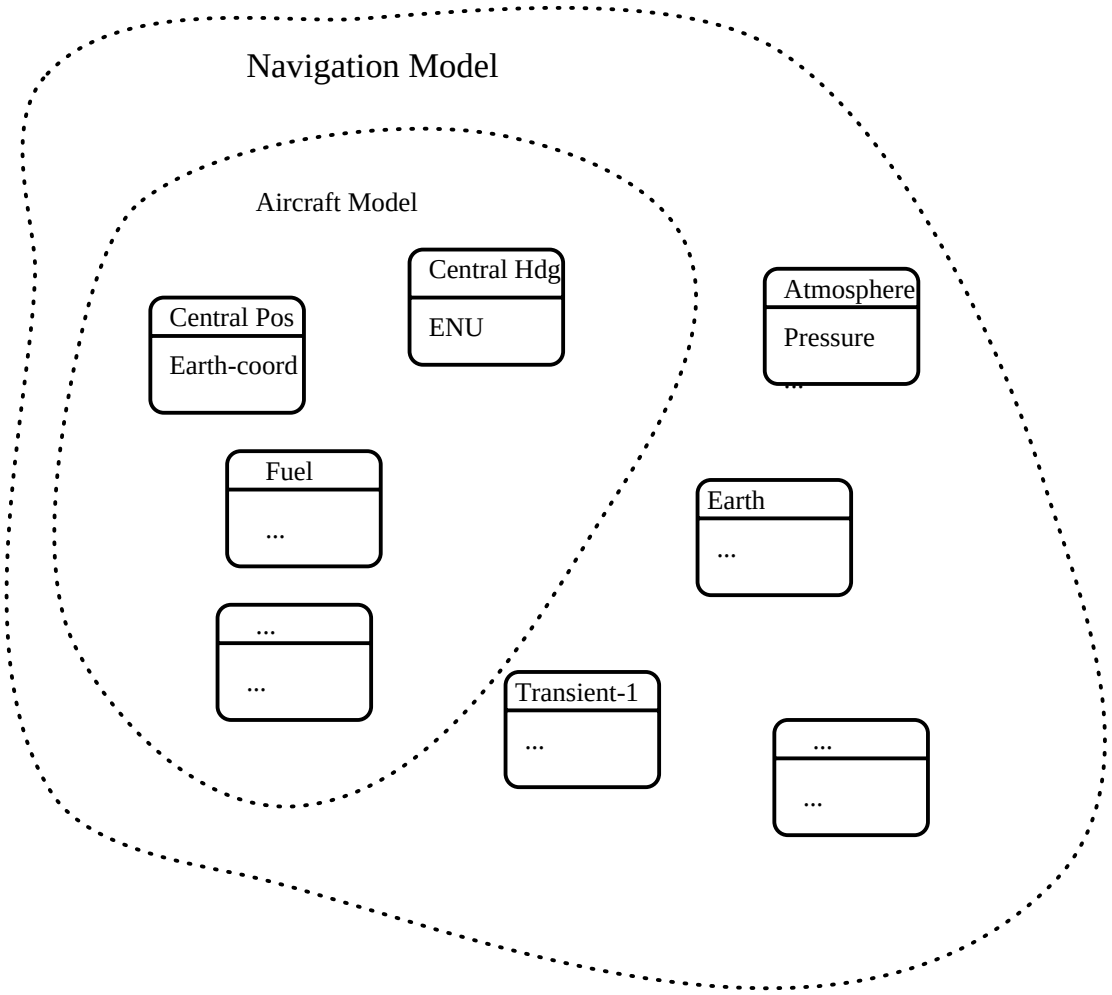
8. Construct an Update (§16) strategy for each attribute in accord with protocols established by Information Flow (§14) policies.

9. Enable construction of Dimensionless Views (§2.3) of values for generic Filter (§12) and signal processing functions when needed to perform updates.

These steps and concerns are applied in different ways across the different kinds of models found in an ACS: Navigation Models (§4), Objective Models (§5), Error Models (§6), Action Models (§7), and Intermediate Models (§8).

4 Navigation Models

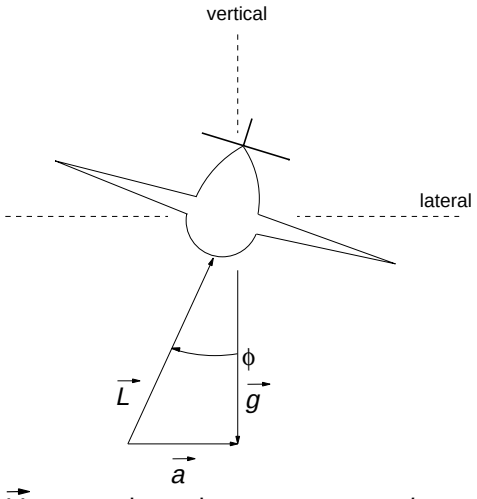
For purposes of avionics control, it's been found that the world of air navigation may be broken up into a few standard categories, with details that vary according to particular ''top-down'' properties needed to support guidance algorithms and ''bottom-up'' capabilities of sensor hardware. Navigation model properties may be needed at a number of points in an ACS, always including the computation of Error Models (§6) and, in many cases Guidance Mode Subsystems (§13), where actual states are used as as initialization or correction data for desired state computation.



Design Steps

Organize navigation models around the four basic categories of the *Aircraft*, the *Earth*, the *Atmosphere*, and *Transient Objects*. Modeling the physical and control state of the Aircraft and its components is of course central to navigation. The geometric properties of the earth and fluid properties of the atmosphere must be represented because some navigation estimates are defined and/or computed relative to them. ''Transient'' objects include other aircraft, obstacles, and radio bases that are noticed and forgotten over time. All model components share a common structure, but necessarily differ in role and usage.

Aircraft. Some values represented by aircraft models are shown in the accompanying diagrams.



$\vec{g}$ = gravity vector

$\vec{L}$ = lift vector

$\vec{a}$ = lateral acceleration vector

ϕ = bank angle

$\vec{V}_g$ = groundspeed vector

$\vec{V}_W$ = wind speed vector

$\vec{V}_{TAS}$ = true airspeed vector

$\vec{V}_E$ = earthspeed vector

V_V = vertical velocity

γ_A = air-mass flight path angle

γ_T = earth-referenced flight path angle

α = angle of attack

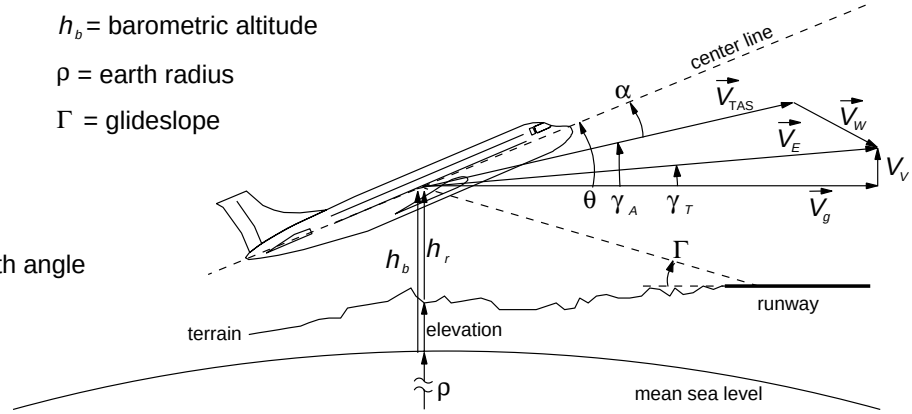
θ = pitch

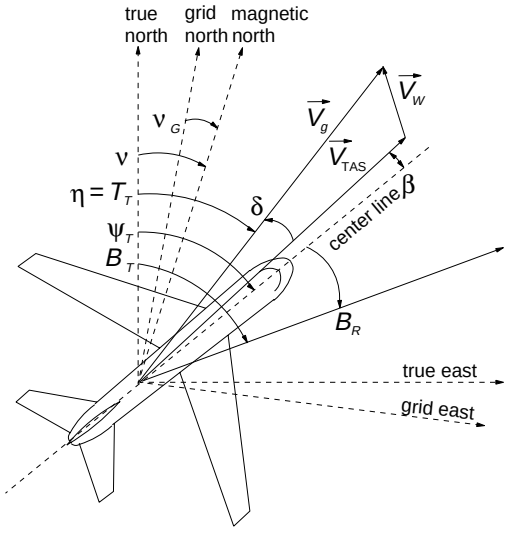
h_r = radar altitude

h_b = barometric altitude

ρ = earth radius

Γ = glideslope





$\vec{V}_g$ = groundspeed vector

$\vec{V}_W$ = wind speed vector

$\vec{V}_{TAS}$ = true airspeed vector

B_T = true bearing

T_T = true track

η = ground track angle

B_R = relative bearing

β = sideslip angle

δ = drift angle

ψ_T = true heading

ν = magnetic variation

ν_G = grivation

The collection of submodels representing an Aircraft forms a classic whole-part structure, paralleling the design of the aircraft itself. For the purposes of an ACS, an Aircraft contains composite attributes such as the overall position and heading, as well as a collection of component states associated with different parts of the aircraft (engines, rotors, wings, wheels, etc). But, as with most ACS models, there is no good reason for making a fixed, explicit model of the Aircraft as a single composite entity. Instead:

1. Organize aircraft state into a group of independent submodels according to the workings and attributes of each part.
2. Collect only those attributes common to the aircraft as a whole into components representing Central Aircraft State. This may itself be structured as a loosely coupled bundle of components, each maintaining a set of independent attributes or views.
3. Aircraft subcomponent models most often play the role of estimation Stages (§8), with data that are collected and combined in assessing central state. They should be structured and used accordingly. Associated Connections (§16.2) need be arranged only across subcomponent models needed to collect and reduce aggregate values across parts into single readings.

Earth and Atmosphere. While the earth and sky are pretty fascinating objects, to a navigation system, they are dreadfully boring. The main reason for modeling the earth is to make sure that the aircraft gets to where it is going, which is normally expressed as a relative position with respect to the earth. The atmosphere is modeled for similar reasons; for example to determine true speed given relative speed with respect to the air.

Earth and atmosphere models incorporate large amounts of static fixed knowledge about geometry and atmospherics. Models of the earth, especially, should be highly pre-wired with facts about the earth geometry, magnetic variation, etc. Updates (§16) mainly provide fine tuning. In most other ways, these models play the role of intermediate Stages (§8), with data that are used as inputs along with others to ultimately estimate useful aircraft properties.

Transient Models. Only a few other real-world objects are of interest to a typical ACS. Some objects, such as Radio Bases, are statically known to exist, but are not always detectable because they enter and leave detection range. Updates and information from such objects usually form parts of earth models. Normally, all other objects (obstacles, threats, other aircraft) need be represented only with respect to fixed type categories and estimates of position and motion.

5 Objective Models

An Objective model represents the desired state of the aircraft. Objective model state values are computed by any number of Guidance Mode (§13) components. These values are compared to estimated actual state represented in Navigation Models (§4) by Guidance Comparator (§12) components in order to Effect (§11) flight.

Although they are conceptually very different, the structure and dynamics of Objective models must parallel that of principal navigation models (§4) so that Error Models (§6) may reflect their differences in a consistent fashion.

Design Steps

Represent the visible attributes of topmost Objective Models as state vectors, using the same, or Transformable (§12), representations used in Navigation Models (§4). While the Flow (§14) of information in maintaining objective model state must parallel that for navigation models, most of the details are different. Rather than estimating values from sensors, objective models use Update (§16) protocols with Filters (§12) that select and/or combine values from various connected (§16.2) Guidance Mode (§13) components, that serve the same role as staged (§8) estimators in navigation. Often, the set of *armed* (enabled) modes established by system Monitor and Control components (§15) provide non-overlapping state data (e.g., lateral position versus altitude), so only a single source will be armed for any single state value. As with estimation, there may be an arbitrary number of intermediate stages (§8).

6 Error Models

An Error Model maintains differences between desired state represented by Objective models (§5) and actual state represented by navigation models (§4). Error Models representing these differences in a form allowing computation of Action Models (§7), in turn to direct flight. The main functional role of comparator components is data categorization and reduction. There are numerous ways in which actual and desired state may differ. These differences must be reduced to an Error Model containing small number of attributes that are meaningful to flight director components.

Most existing algorithms for computing these quantities each specialize on only one error dimension (e.g., speed). Thus the Error model and guidance comparator system cannot be designed as a single monolithic entity. Comparators must be constructed as a small collection of problem-solving specialists, each reducing actual versus desired state differences to one, or perhaps a few Error Model attributes.

Design Steps

Maintain only those attributes useful in computing Action Model (§7) components. In existing ACS systems, these are typically represented by values representing deviations from desired state on the three dimensions of *vertical* (altitude) *error*, *lateral* (heading) *error*, and *longitudinal* (speed) *error*.

Especially when Action Models are themselves specialized along these dimensions, the error model may take the form of a bundle of tiny independent models, each maintaining representations along only one of these dimensions. Update (§16) error models using Guidance Comparator (§12) components.

7 Action Models

Action models (also known as Flight Directors) are the top-most models in an ACS, ultimately rooted in Sensor (§10) data and flight Objectives (§5). Action models are used to update displays and manipulate Effectors (§11).

Usually, each action model corresponds to an error model, spanning the dimensions of dimensions of *vertical*, *lateral* and *longitudinal* control, but where *Terrain following* is dealt with independently. Values that are needed in order to direct flight typically include:

Lateral: Heading Error, Heading Error Gain, Roll, Roll limit, Roll Gain, Lateral Error Scale Factor, Trim Roll.

Longitudinal: Speed Error, Pitch, Speed Error Gain, Pitch Limit, Speed Error Limit, Longitudinal Cue Gain, Longitudinal Cue Error Scale Factor, Pitch Cue Limit.

Vertical: Altitude Error, Pitch, Roll, Vertical Velocity Lead, Vertical Velocity, Collective Position, Altitude Error Gain, Pitch Gain, Roll Gain, Altitude Error Limit, Coupled vertical command limit, Vertical Velocity Coupled Gain, Vertical Velocity Gain, Collective position gain, Collective position error scale factor, Cue Limit.

Terrain Following: g-command, Collective Position, g-command gain, Coupled vertical command limit, Vertical Velocity Coupled Gain, Collective position gain, Collective position error scale factor, Cue Limit.

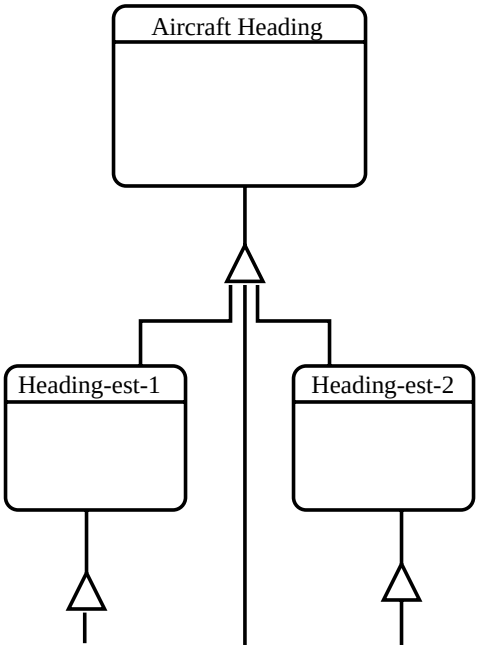
In Avionics Control Systems, most actions are represented as sets of desired effector settings (on/off values, categorical choices, numerical levels, etc) of mechanical components, tailored to the particular kinds of effectors contained on the aircraft.

Design Steps

Maintain attributes representing desired effector settings, in a form useful for Effecting (§11) flight. Partition common logical effector settings into common groupings, normally resulting in a bundle of submodels rather than a single action model. In addition to Error Model (§6) values, arrange inputs from Navigation Models (§4), from Monitoring and Control (§15), and/or directly from simple Sensors (§10) to determine if effectors are enabled and available for manipulation, and invoke associated Action Functions (§12) to compute associated action values and Update (§16) attributes.

8 Staging

The four kinds of ACS models maintain the four kinds of stable representations needed along the course of system flow. But each is only the ‘‘top’’ of series of representations of different aspects of the world derived from sensors and other inputs. There may be several intermediate representations of state vector values in between sources and best possible estimates. Also, many submodels of primary models are used only for purposes of estimating aggregate values in others, so mainly serve as intermediate estimators.



Whenever it is necessary to maintain intermediate representations, *Estimator* components should be constructed. If they take the same form as the main models themselves, architectures assume an extensible, recursive pattern in which outputs of one stage become inputs for the next. This also permits code-sharing of implementations of common features of estimators and top-most models.

These considerations may also be applied to ‘‘mandatory’’ models. Sometimes even these don’t really need stable representations. In this case, the steps may be applied in reverse: State vectors may be passed around through filters as data values without ever ‘‘resting’’ in a model.

Design Steps

Determine the numbers and kinds of estimator stages that are needed, by considering:

- The amount of Filtering and Transformation (§12) needed between sources and principal representations.
- Whether stable representations of these intermediate estimates are needed. For example, if values are used for multiple purposes, or if multiple physically redundant components must be used for the sake of fault tolerance, or if historical values must be preserved, or special actions must occur upon each update. Otherwise, these intermediate values may be maintained implicitly and transiently as inputs and outputs of composite Filters (§12).

Each of these estimators can then be structured (§3), connected (§16.2), updated (§16) and controlled (§14) in the same fashion as top-most models.

9 Sources and Sinks

There are three kinds of external connections and devices in an ACS:

- Sensors (§10) that obtain raw observations about the state of the world used to Update (§16) Navigation Models (§4).
- Effectors (§11) that manipulate mechanical components in accord with Action Models (§7).
- User interfaces and instruments that display suggested Actions (§7), Flight Objectives (§5), and the State of the ACS (§15), and accept pilot input for changing Modes (§13), and Arming (§2.2) components.

Most source and sink components in an ACS must be designed and constructed independently from the models and transformations that interact with them. This is because, in general, there is a many-to-many relationship between devices and other components:

- A single input device may provide data used to estimate many different model properties.
- Many different input devices may provide data about the same property.
- One output device may collect, display, or use information from several models.
- A single effector setting may be transduced through several output devices (e.g., in the case of redundant hardware and parallel paths that simultaneously display and perform actions).

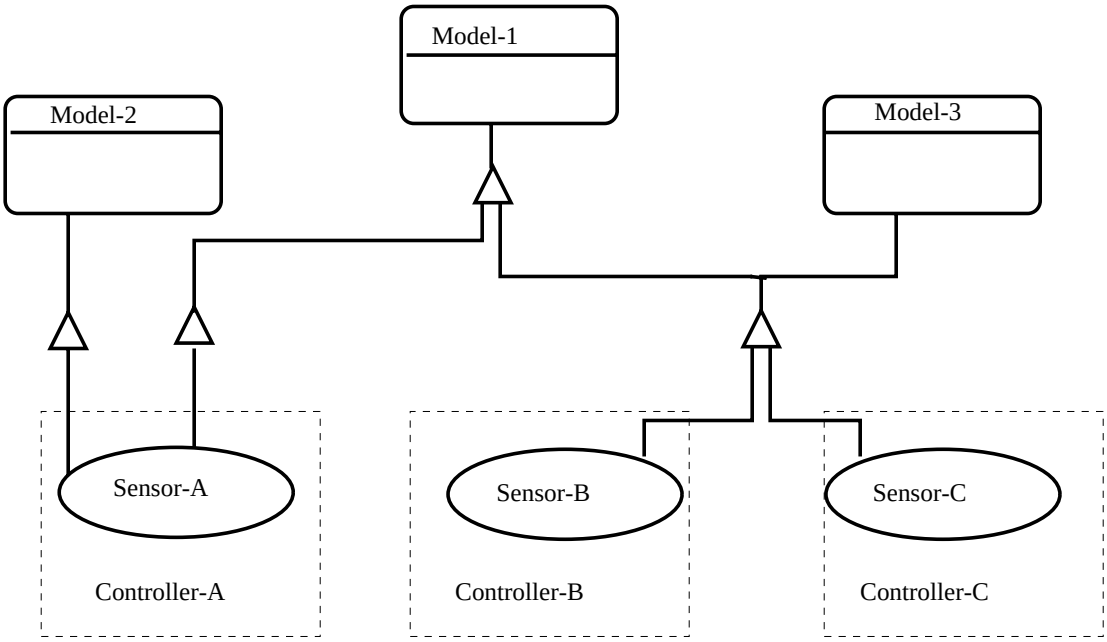
Design Steps

Separate device control from information flow by constructing Device Controllers. Sensor (§10) device controllers differ in most details from Effector (§11) device controllers. The control of inputs and outputs is generally device-specific, while the overall ACS Information Flow (§14) is generally estimate-specific. User interfaces and instruments are special kinds of input sensors and output effectors. (The design of user interfaces for avionics systems is not otherwise addressed in this set of patterns.)

10 Sensors

A sensor is any component that feeds raw data values into the ACS for purposes of model estimation. There are multiple ways of sensing and estimating just about every physical attribute of the earth, atmosphere, and aircraft. Physical sensors that provide the raw data vary in quality, reliability, and the extent to which their values must be filtered and combined with others to obtain useful estimates. Survivability and fault-tolerance constraints require that the system use whatever data is actually available; the ACS cannot shut down just because a sensor goes out of service.

Sensors form the bottommost elements of Staged (§8) model Updates (§16). A sensor might be as simple as a software switch recording the fact that a pilot turned on an instrument or as complex as an Air Data Computer that reports a number of atmospheric measurements. Some user inputs to the system (e.g., manually entered position coordinates) also play the role of sensor data. Sensors may even be simulated by software during development and testing. A *detector* is a special kind or role of sensor. While a normal sensor provides updates for stable models, a detector initiates the process of classifying and describing a new Transient Model Object (§4).



Design Steps

To separate the ‘‘physical’’ control of sensors from the use of their information, for each kind of sensor device, associate a Sensor Controller that handles the special control features of the device and its interactions with the environment. Standardize operations across devices as much as possible, to avoid coupling device-dependent actions to components that deal with their data as estimate sources:

1. Standardize upon common interfaces for Sensor Controller operations. These may include operations to initialize, to shut down, to self-test, to begin a reading, to supply correction data, to enable internal pre-filtering (which may lead to further interactions with models and filters), and so on. For the sake of uniform client interaction, it is worth the trade-off to pad these interfaces with operations that are only sensible for certain subtypes of devices, so long as they can be

implemented as no-ops for others. Despite such attempts to standardize behavior, sensor devices remain notoriously idiosyncratic. It is most likely that some Controllers will need special care and treatment, requiring special code that may infiltrate even far-removed components that somehow rely on their data values. Positions midway between full standardization and full chaos are also available. For example, it may be the case that all Inertial Navigation Sensors (INS) or all Radio Navigation Subsystems can share the same or subtyped interfaces.

2. Because there may be multiple consumers of data, unrelated components may need to discover the state caused by the actions of others. To enable this, device control state must be exposed and standardized into a common set of attributes and accessors. Data clients then may determine whether and how data values may be obtained.

3. Specialize general Update (§16) protocols for sensors by ignoring aspects dealing with data sources except in those cases where the devices do in fact need data from other sources in order to obtain readings.

4. Adopt Monitoring and Control (§15) policies, components, and operations to deal with device control; for example recalibrating when Accuracy (§2.2) estimates fall below thresholds. In order to avoid contention, inconsistency, protocol failures and non-determinism in control state, implement the associated operations to be insensitive to redundant calls by unrelated clients, and contain rules for dealing with inconsistent commands. Alternatively, or in addition, define a secondary controller layer that performs this layer of bullet-proofing, weeding out commands, locking access, and caching consistent sets of data values. Such mechanics avoid the need to synchronize the actions of otherwise independent data clients with respect to each other.

11 Effectors

Action Models (§7) partition common logical effector settings into sets of attributes. Individual effectors may be separately enabled or disabled by Monitoring and Control (§15) components. However, when jointly enabled, all effectors must be consistent (i.e., all must be performing and/or displaying the same action.)

Design Steps

Construct per-device Effector components, that perform actions in accord with these attributes in either of two ways:

Advice. User interfaces or instruments that display Action model attributes in a manner useful to pilots.

Transduction. Devices that transform action representations into direct manipulation of mechanical components.

Structure effector Update (§16) protocols with respect to Connected (§16.2) action models in a manner guaranteeing Consistency (§16.3) across multiple devices that must reflect the same actions.

Mechanical effectors may fail or change state in unpredictable ways. Also, pilots may be able to intervene and override mechanical controls at any time. Most effector devices possess feedback capabilities to deal with such events. Thus, nearly every effector component contains special kinds of Sensors (§10) and even stripped-down Models (§3) to represent internal state, serving as data sources to Monitors (§15) and other ACS components, normally resulting in interactions with high-priority pilot displays, Action Functions (§12), alternative Guidance Modes (§13), and effectors to deal with failure. Support these feedback and status operations in the same fashion as Sensors (§10).

12 Filters

Filters, or more accurately, ‘‘Transformers’’ form the functional glue between Sensors (§10), Models (§3) and other intermediate Stages (§8), and Effectors (§11). But the algorithmic properties of most filter functions are independent of the entities using them. Filter functions do not model the world; they are instead methods for transforming data. The same function may be used in Updating (§16) several independent Models (§3). Thus, most filter functions cannot be designed in ways that are intrinsically coupled to any particular source or destination components.

Classically, a filter is a pure function that accepts one or more main inputs, and returns an output of the same form as the main inputs, but with different values. However, in practice, filters take a number of functional forms, including those that also perform transformations and conversions, and those that include control or parameterization arguments in addition to their main inputs. And while filters are conceptually defined as pure stateless functions, they may be implemented as operations of statefull components that maintain values across different operations to cache recurring computations, and maintain static data, policies, and/or rule sets guiding transformation. Categories include:

Null. Null filters are convenient ways to represent the lack of transformation when updates consist only of simple operations that do not require invocation of an external filter. For example, in the case of single source faultless data (e.g., from a sensor that records a Pilot choice on an instrument) updates may collapse to a single value assignment.

Reducers. Collect and combine a series of values to a single estimate. Reducers may apply weights to various inputs, average multiple estimates, merge redundant readings, and so on.

Selectors. Select and output the best of multiple inputs according to certain rules or criteria.

Weighted. Weight and combine multiple inputs based on their Accuracy and Quality (§2.2).

Sequential Filters. Accept a series of inputs representing values observed over time, and smooth them, average out transient noise, etc., to generate a single best current estimate.

Differencers. Transform two main inputs to a single value representing differences between them. Sequential versions detect rates of changes of inputs.

Converters. Scale or transform Units (§2.1) or types of representation. Converters are required in cases where the same information is represented in multiple formats. For example, there are many ways of representing Earth Geometry, and clever algorithms best suited to each. Even the simple notion of Position can be represented in many ways, relative to different coordinate systems, so values must ultimately pass through converters when unified with other representations.

Sequentially Composed. One filter may further coordinate the effects of others by sequentially connecting the outputs of one or more function as inputs of others.

Conditionally Composed. Invoke best available function depending on Accuracy and Quality (§2.2) estimates of source data, and the preconditions or accuracy profiles of possible transforms.

Dynamically Composed. The filter is an operation of a component that contains an updatable invocation plan consisting of handles to available subfilters and rules for how and when to invoke them on its main input.

Guidance Comparator. A special kind of differencer comparing desired state represented by Objective models (§5) and actual state represented by navigation models (§4), and using them to help produce an Error Model (§6) representing these differences in a form allowing computation of Action Models (§7). Comparator components mainly perform data categorization and reduction. There are numerous ways in which actual and desired state may differ. These differences must be reduced to an Error Model containing small number of attributes that are meaningful to flight director components. Current algorithms for computing these quantities each specialize on only one error dimension (e.g., speed). Thus the Error model and guidance comparator system cannot be designed as a single monolithic entity. Comparators must be constructed as a small collection of problem-solving specialists, each reducing actual versus desired state differences to one, or perhaps a few Error Model attributes. The inner workings of comparators are dictated by algorithmic concerns, and oftem employ specially crafted algorithms.

Action Control. An Action Control Function ''converts'' differences between actual and desired state as represented in Error Models (§6) into values representing actions that will minimize these differences. Algorithms are based on ''control laws'' that are usually specialized along a single control dimension; for example those that map heading error into a set of effector values that cause the aircraft to roll. Different action functions, or different parameters of the same functions may be constructed to reflect control and availability information. For example, a special function may be needed to compute the best course of action if an engine fails.

In Avionics Control Systems, such functions are used extensively to derive attribute estimates from raw sensor data and to combine multiple estimates. The algorithms must be derived from analytic study of device and value characteristics, which are not addressed in this set of patterns. However, even assuming that the best signal processing algorithms are available, there is still the question of how to build ACS systems using them. Usually, for theoretical, empirical and/or and operational reasons, only certain combinations of source types, filter types, and destination types can be used together. This set of compatibilities constrains the ways in which components can be connected, so must be collected to guide Update (§16) mechanics.

Design Steps

1. Organize filters as an extensible set of free-standing procedures and/or classes.

2. Minimize the number and dependencies of filter function *types* by expressing arguments and results as Dimensionless View Types (§2.3) whenever possible, and standardizing on their order and meaning. Some filters, especially Guidance comparators and Control filters are usually too specialized and quirky to standardize upon. However their internal structure may be composed of parts with more uniform characteristics.

3. Standardize on a common method for each filter function for which output quality is a nontrivial function of input quality, to produce in addition to its main output, a means of indicating the assessed Quality and Accuracy (§2.2) of this result, typically as a function of the quality of inputs. Similarly, standardize on a method for callers to determine value or quality preconditions for using any filter that requires inputs be of a certain quality even to be invoked.

4. Ensure that filters may be combined; that their outputs are usable as inputs for other filters. When a specific composition of filtering is needed in Update (§16) operations, define a new filter function

that provides this transformation by invoking others. For the sake of uniformity, it is useful (but not always essential) to encapsulate those composite filters that are actually used in a system as separate free-standing components.

5. Organize large-granularity filter components such as guidance comparators and action control filters as networks of simple filters Connecting (§16.2) an arbitrary number of intermediate Stages (§8).

6. To allow the use of mixed representations, construct a matrix of feasible converters that accept measurements, state vectors, and/or model types, and produce others that are equivalent at the desired unit of measure, that may then be used in a dimensionless manner in a filter function. This matrix may even be directly represented and used as a key into available converters.

7. Statically represent (perhaps only as a design-time tool) a set of feasible connections between sources, destinations, and filters, at the level of *types* (interfaces, state vectors, function types), not individual instances. Because they are based ultimately on the characteristics of available representations, sensors and filters, compatibility constraints are nearly always fixed at design time. When types form an inheritance hierarchy, this set of constraints may be represented more economically. For example, if the kind of positional data from *any* Inertial Sensor may be used with a particular kind of filter type to produce a better positional estimate, then special versions representing special kinds of Inertial Sensors need not be listed.

13 Guidance Modes

A Guidance Mode subsystem is a specialized component that computes the desired state of an aircraft according to a particular set of rules. Examples include Direct-to-point, Course-to-point, Taut-line, and Terrain Following. Even though Avionics Control Systems often contain several Guidance Mode subsystems, usually only a few of them are enabled by system Monitor and Control components (§15) at any given time. Desired state values from different modes are selected or combined in forming principal Objective Models (§5).

Guidance Mode subsystems are special kinds of *planners*. Planners intrinsically possess self-similarities with the systems they deal with, but at a simpler, more specialized level: They represent facts about the world (e.g., properties of courses in a course-to-point subsystem), policies about sets of desired properties (e.g., efficient, legal courses), along with rules and algorithms for attaining them (e.g., staying on course).

To compute desired state, most guidance mode subsystems require input values indicating actual state. For example, a course-to-point mode uses initial position values, along with flight objective information including final position values to chart or find an appropriate course in accord with statically represented knowledge of courses and algorithms for following them. Updates of actual position may be needed periodically as correction data to recompute courses.

Design Steps

Structure Guidance Mode subsystems as miniature open-ended specializations of the ACS System (§1) in which they are embedded. So long as they are of the appropriate general form, most details are unconstrained and entirely arbitrary from the point of view of the rest of the ACS. Typical features include ‘‘lighter’’ versions and variants of main ACS components:

- Representations of flight objectives obtained from components in the style of Sensors (§10) that obtain inputs from users and/or Monitoring and Control (§15) components that manage flight objectives, as well as enable, suspend, restart, etc., the subsystem.
- Specialized static knowledge, or ways to obtain this knowledge (e.g., from radio navigation components) about aspects of the world needed to compute plans. This may be structured in the same fashion as static knowledge representations in principal ACS Models (§3).
- Representations of actual position, obtained when needed (perhaps only on initialization) from Navigation models (§4).
- A set of rules, of the same general form as navigation Filters (§12) that compute desired state given the results of Comparators (§12) between predictions of the results of maintaining current state versus objectives.
- Interfaces, operations, and protocols allowing Updates (§16) by principal Objective Models (§5).

14 Flow

Information of various sorts in an ACS is carried ''upward'' from Sensors (§10), through Navigation Models (§4), to Error Models (§6) to Action Models (§7) and Direction (§11). However, the *control* of this flow can be assigned in several ways among sources, intermediaries, and consumers.

For any pair of source and consumer components in a system, there are three basic strategies (plus countless combinations and variants) available for moving transformed information between the source and consumer[10]:

Pull (Functional) The consumer requests information via a procedure call to the source that returns the values as results. This operation may be implemented via an arbitrarily complex sequential protocol, may be multithreaded with other requests on either side, and may perform ''in-place'' updates rather than returning results.

Push (Event Driven) The source issues an message with data values as arguments whenever new values are available. Push-style models may be implemented as resultless procedure calls containing new data as arguments, as non-returning point-to-point messages or broadcasts, as prioritized interrupts, or even as continuation-style program jumps.

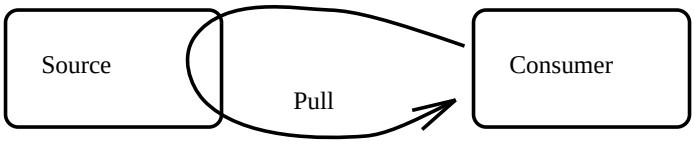
Indirect (Shared Resources) The source periodically makes data available in known locations for the consumer to read. This is often implemented via transfers to shared memory occurring at a fixed known periodicity, or via polling of synchronization signals. It may also be implemented in a more active ''blackboard'' style, where data producers asynchronously push information to a managed, synchronized common resource, and data consumers pull it.

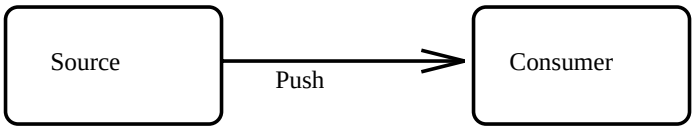
Because these matters affect every pair of communicating components in the system, because nearly all mechanisms must preserve the overall system flow requirements in a consistent fashion, and because mismatches in modes are difficult and overhead-consuming to patch together, control conventions must be established by *policy*.

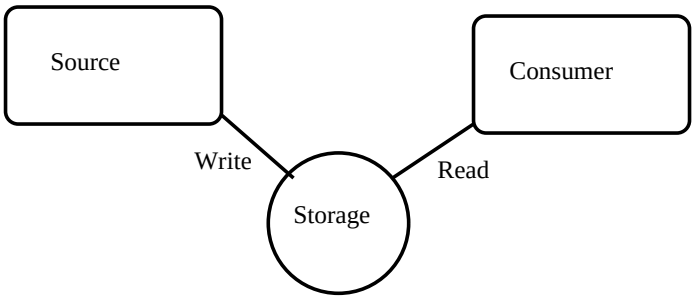
Sets of flow policies help generate the overall architectural style of a system. Different policies lead to substantial differences in interfaces and protocols of staged components, which in turn affects their use and reuse across systems. For example, in most pull protocols, navigation models need to record and invoke Connections (§16.2) to sources, each of which requires data-access operations. In most pull protocols, sources must record connections, and consumers must support operations triggering updates.

The style of interaction of certain components is often beyond the control of system developers. For example, some Sensors (§10) can only operate in indirect mode, via periodic direct memory access (DMA) transfers. Similarly, components that will be Configured (§17) to reside on different processors may be subject to communications infrastructure constraints that make push protocols the only reasonable choice. And although signal processing Filter (§12) functions may be invoked by components otherwise obeying any of these protocols, the filter procedures themselves almost always take a functional, pull-driven form. Finally, system Monitoring and Control (§15) operations often work in the reverse direction of primary information flow, and must use reverse protocol styles.

Thus, no single option is always best or even attainable. Moreover, each can pose problems. For example, pull protocols cannot deal with unexpected or prioritized events. Push protocols cannot always be







buffered effectively, and require the existence or development of scheduling policies and algorithms. Shared memory requires the overhead of synchronization to avoid storage inconsistencies.

Design Steps

For the sake of minimizing design complexity and maximizing configurability and performance, adopt the smallest possible number of global default policies, each perhaps with a hand-full of special-case exceptions. Map out where each policy applies.

Center information flow policies on Navigation Models (§4), the components that maintain snapshots of the world. These components are at the point of maximum stress, and bear most responsibility for ensuring that the right values are correctly Updated (§16) and propagated.

When they are not prespecified, develop the infrastructure components, subsystems, and hardware necessary to support policies. These may include schedulers, priority queues, and event handlers for managing push-style communication, remote procedure calls and thread libraries for pull-style communication across processes, and memory management mechanisms and synchronization primitives for indirect-style interaction.

Examine existing similar systems and existing reusable components for guidance in establishing policies.

For example, in several ACS systems, data are obtained using hardware-assisted periodic shared-memory input from external sensors. Every N time units, processing stops while sensors transfer readings. After a suitable settling period, other components use this data. On the other side, most updates are pull-driven from guidance. This global synchronization technique avoids difficult local synchronization problems that would otherwise occur with shared memory, as well as those that would be encountered when trying to integrate, for example, readings from one push-style sensor with that of a pull-style one. However, there are still exceptional push-driven flow paths, for example to deal with high-priority matters such as obstacle avoidance.

15 Monitoring and Control

Monitoring and control components supervise ACS functionality by:

- Collecting inputs from other components to monitor their functional state.
- Logically and / or mechanically constructing, initializing, enabling, resetting, and / or disabling ACS components and devices.
- Displaying status and accepting new control instructions by pilots.

The most important consideration in designing the supervisory structure in an ACS is the degree of *centralization* needed or desired to perform requisite actions. In a fully centralized approach, a single system supervisor continuously assesses the state of every component whose reliability may vary, and and manages the state of every component whose behavior may be altered. This leads to simple control mechanics, but can generate bottlenecks and unacceptable performance. In a fully decentralized approach, each component assesses and manages its own control state, as well as that of other Connected (§16.2) components in the course of its primary operations. This avoids the need for supervision, but requires often-unattainably accurate and complex synchronization and propagation operations to nearly every component in the system.

Hierarchical approaches represent the best and most common middle ground. Here, each component manages strictly local concerns. Intermediate supervisory components manage groups of components, but are in turn managed by more centralized supervisors. Taking a hierarchical approach hits the worst aspects of full centralization or full decentralization only when they unavoidable due to the nature of the particular supervisory problem at hand.

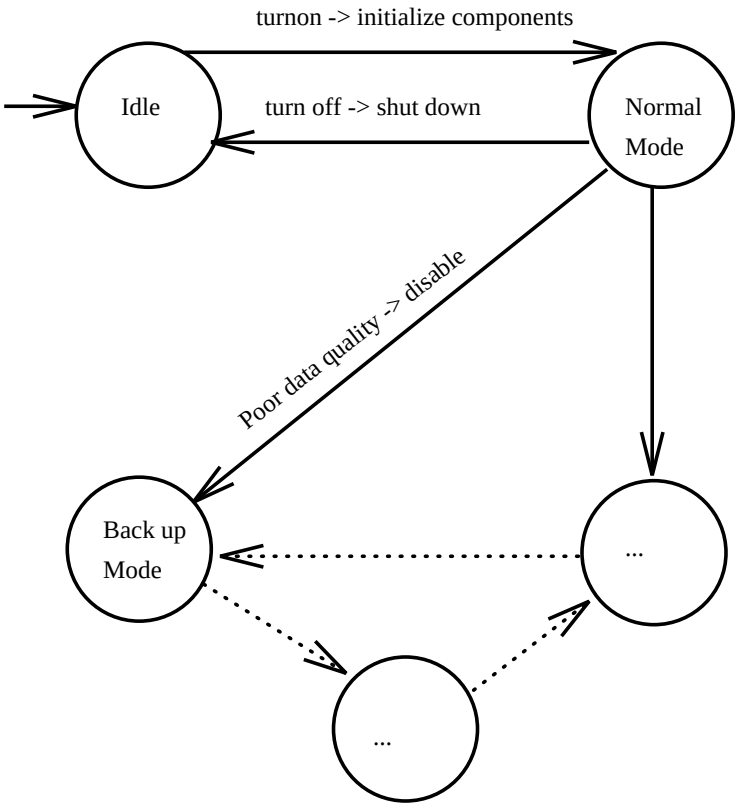
Design Steps

Organize monitoring and control facilities into hierarchically partitioned domains. When possible, match this partitioning to Configuration (§17) strategies so that control domains mirror the implementation-level decomposition of the system.

For each, define a *Supervisor* that interacts with and manage states of components under its domain. Supervisors are special kinds of *state machines.* Their states reflect in-the-small Models (§3) of selected aspects of the system itself, and their transitions result in actions that change the state of components under their domain.

Supervisors may be constructed using standard methods[2, 4, 11] for designing and building state machines:

- States represent the status of one or more components.
- Inputs represent changes in status of components, obtained using an agreed upon notification Protocol (§14).
- Transitions represent the necessary or best consequence of a component state change, reflecting policies and constraints among allowed choices of states and modes.



- Outputs on transitions are commands that actually alter the states of components, construct and initialize new components, and resume or suspend modes of operation. When commands cannot take effect instantaneously, adopt policies and synchronization mechanisms to ensure consistency; for example, that changes always take effect in the process of next Update (§16) of a Model (§3) or perhaps all of them within the control domain.
- When appropriate, tie inputs and outputs to user interfaces, accepting inputs from pilots as well as internal components, and channeling outputs to status displays. For example, the top-most supervisor components must turn on the ACS as a whole when so instructed.

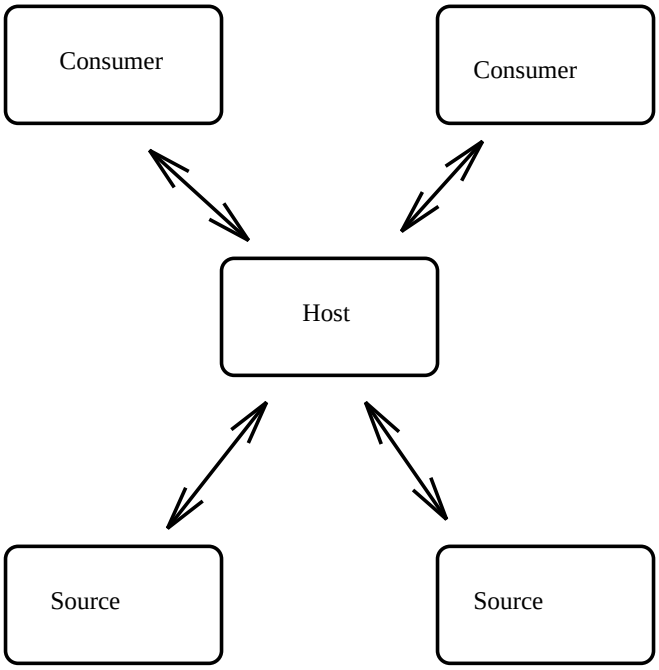
On the supervisee side, monitoring and control operations conceptually sit atop those dealing with the overall functional Flow (§14) of an ACS. Ideally, monitoring would simply represent additional independent ‘‘outputs’’ performed in the course of model Updates (§16), and control would represent independent inputs. In principle, these should be layerable on top of base functionality. However, in most cases in practice, they must be spliced directly into base operations, adding complexity to both representations and algorithms. In particular, since the vast majority of supervisory operations in an ACS revolve around the maintenance of Accurate (§2.2) models, these concerns infiltrate the design of nearly every ACS component.

16 Updates

Avionics Control Systems are in constant flux. New data values become available to Model (§3) components on a continuous basis, allowing them to compute fresh estimates. To maintain required system-wide information Flow (§14), each update from the level of Sensors (§10) to Effectors (§11) should take the same general form.

Design Steps

In a Staged (§8) design, each stage of estimation should have the same essential form and characteristics. The version used in any given ‘‘host’’ component may omit steps and considerations that do not apply given a particular set of Flow (§14) control and Configuration (§17) decisions.



16.1 Protocols

Ensure that each update procedure contains all relevant steps of the following sequence:

1. Determine if the host attribute(s) to be updated are Armed (§2.2). If not, skip all other steps.
2. Determine one or more Armed (§2.2), Connected (§16.2) data sources for the estimate.
3. Obtain data from each source. This may involve, for example, pull-style interactions with a Sensor Controller (§10) or another model component; or a timed wait for a ‘‘push’’ of the data, giving up on time-out; or in a periodic indirect transfer strategy, checking any validity indicators for values that should be present on each update cycle.
4. Determine the assessed Quality (§2.2) of obtained data.

5. Determine one or more Filters (§12) that may be used to transform all obtained source data into a new estimate of a given quality. Special filters and/or internal actions may be chosen when the set of sources changes across updates, so as to ensure a smooth transition in estimates.

6. Invoke the Filter (§12) with available source data, typically in the form of Dimensionless Views (§2.3) of values after scaling to common units.

7. Update attributes from the results of the Filter (§12). If appropriate and possible, first check estimates according to statically encoded rules to determine whether they fall within known acceptable ranges or relations with other values.

8. Update Accuracy (§2.2) estimates. The ''bottom-up'' flow of merit figures parallels that of attribute values. At the bottommost level, Sensors (§10) and/or their direct consumers must maintain independently or inherently known accuracy figures. These may be propagated along update paths, when necessary and possible through extra parameters of Filter (§12) functions that both compute transforms and their quality.

9. For all Connected (§16.2) data consumers of the new estimate, if any action must be taken to propagate values to them, do so. A host may be one of its own consumers, for example when ''derived'' attributes must be computed on the basis of updates to primary values.

Failures. Arrange evasive action upon failure in any of the above steps; i.e., lack of sources, inability to obtain data, insufficient source data quality, lack of an appropriate filter, failure of a filter component, inconsistent or poor quality filter results, or propagation failures. Normally, the most appropriate action is to control Arming (§2.2) of the host component and/or its data sources. This may involve interactions with Monitoring and Control (§15) components that alert them when input values are of insufficient quality to form new estimates (as determined, for example, by comparison to fixed thresholds). Local and/or centralized operations may then disarm, recalibrate, or mechanically shut down Sensors (§10), and propagate top-down disarming control to other data sources.

16.2 Connections

Uniform update strategies work so long as each ''host'' stage can determine the set of dependent data sources and associated data types, the kinds of Filters (§12) to form new estimates, and the set of dependent consumers for which action is required by the host upon each update.

The first step in establishing connections is to determine the set of *dependencies*; that is, cases where any change in one attribute must propagate to another that depends on it. Every set of connections must reflect Compatibility constraints (§12) among filters and representations. However, compatibility constraints operate at the level of types. Connections are at the level of individual components. Each host must deal with the particular data sources and consumers required to fulfill its role in the system.

A range of options for *representing* connections apply to both source-to-host or host-to-consumer connections. Each impacts different aspects of Update (§16) mechanics, and so must be considered in tandem with update protocols. Feasible choices depend upon Flow (§14) and Configuration (§17) strategies.

Anonymous Connections. No explicit representation of source connections is required if the host always waits for ''push'' style data of certain forms to arrive, or for indirect data in pre-established storage

locations to fall into place, without knowing or caring where it came from. When all the right data are obtained, it proceeds with an update. Similarly, there is no need to explicitly represent consumer connections if, upon update, the host either broadcasts data to all possible consumers or places state vectors in pre-established storage locations for use by consumers.

Fixed Connections. The host invokes (or waits for) a fixed set of data-gathering operations before each update, and invokes a different fixed set of operations after update. Sources may include current or previous attribute values of the host itself, perhaps to be averaged with new readings. They may even include higher quality data from ''upstream'' models fed back to be used at this level.

Dynamic Connections. The host contains operations that accept new handles (object references, ''call-back'' function pointers, etc [4]) to source and/or consumer operations to be used on each update. These handles are supplied (by either the sources themselves or Monitors (§15)) when new sources become available, and may be discarded by the host upon noticing connectivity or data Quality (§2.2) problems.

Brokered Connections. Each update is mediated by a relay or broker that maintains association lists between each desired data type and available suppliers.

16.3 Consistency

Multiple or interdependent representations are often essential in Avionics Control Systems in order to apply Filtering and Transformation (§12) algorithms. For example, *position* may need to be represented both ENU form and XYZ form for purposes of a particular set of algorithms. But whenever interdependent representations are used, consistency problems are all but inevitable. Any two state vectors that represent or derive from the same physical phenomenon must be consistent. Changes in one must be reflected in the other.

No approach can or even should guarantee that all representations in any system are always instantaneously consistent. This is especially so in ACS systems, where updates to values continuously stream in, and where it is sometimes better to use an old value rather than undergo the overhead or delays associated with obtaining a new or consistent one. Further, if differences are somehow known to always lie within an acceptable fixed tolerance and they are either never directly compared or are only compared within this tolerance, then you can ignore the entire problem.

Otherwise, update strategies must be modified to deal with sets of interdependent representations; that is, cases where two different state vectors contain the same or interconvertable values representing the same physical phenomena. Two overlapping representations, A and B, form a cycle of dependencies. There is no single technique that applies, but most instances in practice fall under one or more of the following special cases.

- If representation A can be treated as ''primary'', values of representation B may be obtained as ''readonly'' Views (§2.3), where changes are never propagated back to A.
- Representations that are obtained from different sources may be treated as different estimates and combined via special Filters (§12) into a new intermediate Stage (§8) maintaining an estimate that is better than either of the two.

- Components maintaining these representations may contain internal propagation protocols to each other that are somehow guaranteed not to loop forever[4]. For example, if A receives an update request from anyone other than B, it notifies B, but if it receives one from B, then it silently performs an update without calling back B, under the assumption that such messages are only consistency propagations. (Note that this assumption is not necessarily always correct in general.)

17 Configuration

While it is profitable to design Avionics Control Systems in an open, extensible fashion, ultimately an ACS must be configured into a *closed*, self-contained system, due to the following constraints:

- ACS components typically reside on known, fixed, special-purpose hardware, with fixed constraints on the placement of components on particular processors, on required forms of physical redundancy, and so on.
- For any given instance of an ACS, there are typically only a fixed set of available devices, modes, and effectors. Thus the concrete architecture of a particular ACS is governed by the *static* structure of a fixed set of components. Although most components must be initialized, monitored, and sometimes disabled by Monitoring and Control (§15) operations, very few components (often none at all) are dynamically constructed or deleted during execution.
- Most systems use off-the-shelf, large-grained hardware and software components such as guidance modes, that are developed independently according to fixed, non-extensible specifications.
- Performance concerns typically require hard-wired coupling of components to avoid the overhead associated with flexible, extensible communication schemes and protocols.

Design Steps

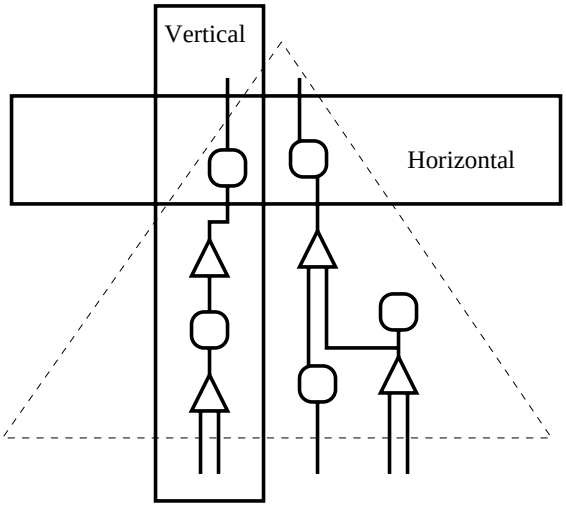
Establish ''wiring'' schemes for composing large-grained *subsystems* in which most internal connections and protocols are ''frozen'', and in a way that the subsystems themselves are suitable for static configuration. Integration into subsystems is by nature an opportunistic task, exploiting relationships among components that lend themselves to tighter coupling. Different strategies have the effect of collapsing overall ACS designs into different-looking concrete architectures, each with different consequences in terms of performance, static structure, and independence. Given a set of components and a configuration strategy, it is possible to build generator tools that can instantiate different versions of a system (see [3]).

There are two nearly independent dimensions to collapsing components. The first is the ''direction'' of integration:

Vertical. Integrate components that deal with an isolated set of model attributes along a subset of a functional Flow (§14) path, even all the way from input Sensors (§10) to Effectors (§11). This kind of vertical collapse applies whenever it is possible to decompose an aspect of functionality into a loosely-coupled, mostly-independent subsystem.

Horizontal. Collect submodels and stages of the same kind of model (Navigation (§4), Objective (§5), Error (§6), and Action (§7)) into larger-grained subsystems. This re-centralizes independently designed model components that operate on independent attributes at the same level of processing.

Vertical integration is more common and usually more desirable. The representations and algorithms used for estimating, for example, positional values based on Radar sensor inputs are separable from those for estimating atmospheric values from Air Data Computers, so may be split off into different subsystems, integrated only at their common meeting point in overall system flow. Similarly, Guidance Comparators



(§12) and associated Error Models (§6) normally work off only a single control dimension that is matched to a corresponding set of Action Functions (§12) and associated Action Models (§7). All components dealing with values along any single dimension can be integrated into a single larger-grained subsystem.

Horizontal collapse can be used to exploit common control Flow (§14) mechanics. For example, all Navigation Models (§4) that work off periodic memory transfer from Sensors (§10) can be collected into a common subsystem with its own scheduler and policies.

The second dimension deals with the ‘‘focus’’ of integration:

Model-based. Directly incorporate functional Filtering (§12) algorithms into Model (§3) Update (§16) procedures. This eliminates the overhead associated with separate construction, normalization of data, and invocation.

Transformation-based. Directly incorporate Model (§3) representations and Update (§16) protocols into Filters (§12). This eliminates the need to maintain representations, and may lead to the collapse of several model Stages (§8) by directly feeding the results of one computation into another without ever explicitly representing the intermediate values.

Model-based and transformation-based strategies are duals (often stemming from object-oriented versus structured-design approaches), that tend to result in about the same final configuration, but expressed in different ways. They most readily apply when representations, algorithms, and/or control are already somewhat coupled, so that this coupling can be accentuated and hard-wired into larger-grained entities. For example, when there is already a one-to-one correspondence between individual Guidance Comparators (§12) and Error Models (§6), it is simple and effective to merge Transformation (§12), Updates (§16), and Representation (§2) of a primary error attribute.

References

[1] Alexander, C., S. Ishikawa, & M. Silverstein, *A Pattern Language*, Oxford University Press, 1977.

[2] Booch, G., *Object-Oriented Analysis and Design*, Benjamin Cummings, 1993.

[3] Coglianese, L., W. Tracz, D. Batory, M. Goodwin, S. Shafer, R. Smith, R. Szymanski, & P. Young *Collected Papers of the Domain-Specific Software Architectures (DSSA) Avionics Domain Application Generation Environment (ADAGE)*, Document ADAGE-IBM-93-09, IBM Federal Sector Company, 1994.

[4] deChampeaux, D, D. Lea, & P. Faure. *Object Oriented System Development*. Addison Wesley, 1993.

[5] Gamma, E., R. Helm, R. Johnson, & J. Vlissides. *Design Patterns*, Addison-Wesley, 1994.

[6] Kayton, M. *Avionics Navigation Systems*, Wiley, 1969

[7] Krasner, G. & S. Pope, ''A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80'', *Journal of Object-Oriented Programming*, August/September 1988.

[8] Lea, D. ''Christopher Alexander: An Introduction for Object-Oriented Designers'', *ACM Software Engineering Notes*, January 1994.

[9] Lea, D., & J. Marlowe, *PSL: Protocols and Pragmatics for Open Systems*, Technical report 94-0369, Sun Microsystems Laboratories, 1994.

[10] OMG, *Common Object Services Specification*, Document 94.1.1, Object Management Group, 1994.

[11] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[12] Roskam, J. *Airplane Flight Dynamics and AUtomatic Flight Controls*, Roskam Aviation, 1979.

[13] Smith, J. M. *Mathematical Modeling and Digital Simulation for Engineers and Scientists*, Wiley, 1977.