A Quasi-Network-Based Fly-by-Wire Simulation Architecture for Teaching Software Engineering

Dan Tappan
Department of Computer Science
Eastern Washington University
Cheney, WA, USA
dtappan@ewu.edu

***Abstract*—Undisciplined cohesion and coupling undermine countless aspects of quality software. Students, however, unfortunately tend to gravitate toward such approaches. This system mitigates this problem by forcing them to communicate with their components through a well-constrained hierarchical virtual network of networks. The application is a dynamic, plug-and-play aircraft fly-by-wire system that processes a wide variety of commands to design, construct, and manipulate sensors, actuators, controllers, and communication buses concurrently in a flexible model-view-controller architecture. It successfully employs many systems-engineering concepts of modeling, simulation, visualization, and analysis toward the goal of instilling disciplined design, implementation, testing, and evaluation practices in students. In particular, it provides the pedagogical and programmatic frameworks for creating, executing, presenting, and analyzing meaningful test cases as part of formal test plans carried out in controlled experiments via scientific method. The system can be adapted relatively easily to countless other real-world multidisciplinary domains for reuse in other projects. Extensive results from classroom deployments show that students overwhelmingly benefit from this approach.**

Keywords—software engineering; control system; simulation

I. Introduction

Software, by its flexible virtual nature, allows programmers to make quick and easy changes, at least compared to related physical disciplines like engineering. Unfortunately, without appropriate self-discipline, it is all too easy to produce messy spaghetti code with endless nasty interconnections. The equivalent rat's nest of wires in an electrical system would be immediately apparent to anyone, but in a software system, no such obvious red flags exist without first learning to recognize them. The goal of this system is to instill disciplined design, implementation, and testing practices in software-engineering students by forcing them to work within a Java model-view-controller architecture that behaves like a well-defined and protected physical plug-and-play network.

The overarching philosophy is to use a systems-engineering approach of modeling, simulation, visualization, and analysis respectively to build a solution, execute it in a controlled way, present the results visually, and analyze what they mean as part of testing and evaluation. The model, in particular, is a simplified aircraft fly-by-wire system that controls a variety of flight components in complex real-time ways. This context, supported by background research, provided students in an undergraduate software-engineering course with a holistic understanding of the problem space such that they could create a corresponding clean implementation in the solution space. The objectives are typical of any software system: to separate the concerns, maximize cohesion, minimize coupling, and delegate responsibility appropriately. These considerations are further complicated by the need for uniform, safe, and repeatable concurrency among the components. Building and testing single-threaded code is difficult enough for most students; multithreaded (or the appearance of such here) absolutely requires a disciplined approach.

II. Background

All control systems take inputs of some sort and produce corresponding outputs of some sort. In traditional (non-electronic) systems, the connections from the former to the latter are mechanical linkages like cables, pushrods, and shafts. For example, a car steering wheel directly drives the gearbox to deflect the front wheels. There is little possible dynamic variation in the operation, like changing the steering sensitivity based on vehicle speed, because the configuration of the static system is fixed. A "by-wire" system, on the other hand, translates the input from a sensor into an electronic signal that travels via a network to an actuator, which acts upon it as output. In this form, any amount of computer processing is now possible for dynamic real-time reconfiguration.

In a fly-by-wire system, the primary control sensors are located in the cockpit in the form of a stick or yoke, as well as pedals, switches, and levers. The actuators are located around the airplane. Fig. 1 shows a basic configuration, which also includes the engines and landing gear.

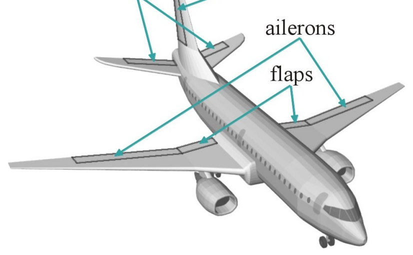

Fig. 1. Basic airplane actuators [1]

This work focuses on the behavior of the control system only, not on its effect on the airplane in flight. In other words, the airplane is basically stationary on a test stand. As a result, its degrees of freedom of motion and aerodynamics are ignored. For background, however, the elevators affect pitch (nose up or down) to change altitude; the ailerons affect wing roll to turn; and the rudder affects yaw to coordinate turns, much like the front wheel of bicycle does.

III. Model

This work uses a model-view-controller architecture. The model is the module that defines the machine being manipulated. The plug-and-play nature of the architecture allows it to accommodate other by-wire systems with relatively little difficulty. Further supporting this goal is the extensive use of well-established, reusable software design patterns [2]. For example, a subsequent offering of the same course used it as the basis of a toolkit for building and controlling heavy construction equipment. Follow-on work is planned for modeling railroads and railway equipment.

A. Datatypes

Significant, consistent anecdotal classroom evidence shows that students have a major problem with abstracting, maintaining, and manipulating data properly. Java primitives are appropriate in earlier low-level courses, but at higher project-based levels like software engineering, they lead to a proliferation of problems. For example, units, magnitudes, and limits are not applied consistently, error handling is inconsistent or almost nonexistent, and code bloats from haphazard attempts at reimplementing similar solutions in multiple places.

To mitigate this situation, the architecture provides a rich set of self-contained concrete datatypes for every kind of relevant data; e.g., Acceleration, AngleHorizontal, AngleVertical, FlapPosition, Identifier, Percent, Power, Rate, Speed, and many dozens more not directly in play in this paper. Each maintains its own error checking and helper methods for manipulating and converting it appropriately. This approach lends itself to convenient unit testing in isolation. It also reduces the burden of documentation; e.g., avoiding having to state and enforce everywhere that horizontal angles are in mathematical degrees (as opposed to navigational degrees or radians) because AngleHorizontal always uses this form.

Another consistent problem students have is with indiscriminate coupling and undisciplined, unprotected sharing of objects. Changing a mutable object in one place may have countless unexpected consequences throughout an entire system. To mitigate this problem, datatypes employ a functional paradigm, which makes them immutable. Any mutable action on them produces a new object via copy-on-write semantics [3]. It is therefore exceedingly difficult for students to interact with the system outside the prescribed network infrastructure, intentionally or not.

Finally, datatypes extensively use Java generics to constrain their application to appropriate contexts. Students are prone to hard-coding dangerous runtime casts and making decisions based on querying objects for their type with the `instanceof` operator instead of properly utilizing the object-oriented principles of inheritance and polymorphism. Explicit casting should be avoided as much as possible in a dynamic, plug-and-play system.

B. Intervals

An interval is the data-structure equivalent of the presumed motive force that moves an actuator from one state to another. The basis is kinematics, or geometry in motion without regard to its causes [4]. Actuators play different roles and therefore have different state types, which the datatypes define, and intervals directly map onto and control.

An interval has implicit or explicit limits; e.g., a Percent interval always ranges inclusively from 0 to 100, whereas a Speed interval ranges inclusively from its specified minimum to maximum values. In all cases, the current state must always reside on the interval. It is impossible for a student to cause an inconsistent state without detection and notification.

Change in state as a delta is also configurable to account for slower or faster movement. In the linear variant, the delta never changes. In the nonlinear, it does so to account for acceleration or deceleration. Again, it is impossible to cause an inconsistent state. The interval always reflects a continuous function; e.g., it cannot achieve maximum delta without accelerating to that value by the rules. Likewise, it cannot change direction without decelerating to zero first. This behavior reflects the reality of the mechanical system that the interval represents.

An interval is like an operating-system process in several ways. A nonpreempting request submitted to it is queued for execution. For example, requesting it to go to maximum value and then immediately to minimum value would entail increasing to completion (with initial acceleration and final deceleration) and then similarly decreasing to completion. A preempting request, on the other hand, would cause the currently executing request to complete gracefully with deceleration to zero as soon as possible, followed by servicing the new request. A terminate request functions in the same way, except that it schedules no subsequent action. A cancel request kills the currently executing request immediately with no graceful shutdown. It is not an option for normal interaction because it violates the kinematics; i.e., infinitely fast deceleration from the equivalent of dividing by zero in

$$rate = distance / time \tag{1}$$

Fig. 2 notionally depicts the state and delta (top and bottom lines of each graph, respectively) for a variety of combinations of linear and nonlinear movement in increasing and decreasing directions with intervening terminate and cancel actions. To demonstrate the value of using the architecture-supported intervals over their own ad hoc implementations, the students had to solve the basic elements of this problem as a standalone proof-of-concept Java program. The results were telling: their solutions were overwhelmingly large, unmanageable hacks, not one worked completely, and the average grade was 11%. They said it was an extremely eye-opening experience and acknowledged the value of the orthogonal approach in this work: one solution applicable to many problems [5].

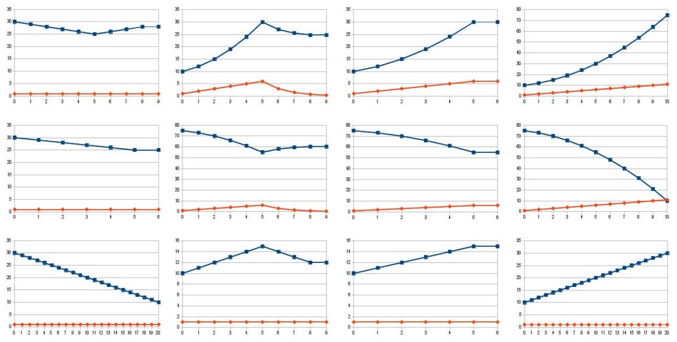
Fig. 2. Sample kinematics intervals

C. Buses

Every interval (as part of an actuator) resides on a communication bus that transfers requests. Three types of interval servicing are possible:

- *Oneshot*: service a request once, then expire automatically; e.g., instantaneous on/off actions.
- *Definite*: service a request until a specified end condition, then expire automatically; e.g., movement.
- *Indefinite*: service a request continuously, ending only upon a terminate or cancel request; e.g., engine rotation.

In addition, requests may specify in detail the timing of the interval servicing:

- *Lead time*: the amount of time that an interval should initially perform no action. This models initialization.
- *Duration*: the total amount of time that the request will be serviced. This applies to definite intervals only.
- *Frequency*: the rate at which actions are performed while being serviced. This allows them to operate at a fraction of the clock speed.

Requests are exactly that: *requests*. They are not imperative commands that must be honored (even though "command" is the conventional term used in flight control) [6]. In fact, the servicer decides how to process the request and responds to the submitter as follows:

- `IGNORED`: ignored and discarded the request because it was not considered applicable.
- `REJECTED_INVALID`: rejected the request, which normally would be serviced, but cannot be now because it is somehow invalid or inappropriate.
- `REJECTED_UNABLE`: rejected the request, which normally would be serviced, but cannot be now for some reason on the servicer's side. The servicer may inform the requester when it is available again.
- `ACCEPTED_BLOCKED`: accepted the request, but it will be queued for later servicing because the servicer is busy.
- `ACCEPTED_SERVICING`: accepted the request, and it will be serviced immediately.

This handshaking approach was difficult for the students to embrace because they are used to shouting at their code imperatively to do what they want, and if it does not, then shouting louder. For example, one student admitted that his development process was to "[keep] throwing more code at the compiler until it shut up."

D. Actuators

Actuators are the interval-based virtual mechanisms that cause the aircraft components in Fig. 1, as well as others, to change state appropriately. The presumed underlying motive force (electrical, hydraulic, pneumatic, thermodynamic, etc.) plays no role, only the resulting action. Section III.F addresses specific actuator behaviors in detail, especially in combination. In general, however, their data (what they are) and control (what they can do) adhere to the following constraints:

- *Rudder*: deflects left or right to an angle.
- *Elevator*: deflects up or down to an angle.
- *Engine*: changes speed as a percentage of maximum revolutions per minute.
- *Aileron*: deflects up or down to an angle; in roll mode, they are always paired antisymmetrically on the wings, so when one deflects, its counterpart deflects by the same amount in the opposite direction.
- *Speed brake*: deflects upward to an angle; on a real aircraft, separate dedicated actuators typically play this role, but for pedagogical reasons, ailerons do so here in speed-brake mode. There is no antisymmetry: all ailerons deflect upward to cause increased drag.
- *Main gear*: extends or retracts as a percentage of downward deployment.
- *Nose gear*: extends or retracts as a percentage of downward deployment, but also simultaneously rotates 90 degrees to stow sideways in the fuselage.
- *Flap*: deflects downward to an angle. As Fig. 3 shows, plain flaps rotate about a fixed point; for double-slotted flaps, this point moves backward, and the flap separates, requiring two coordinated intervals that are both simultaneous and sequential.

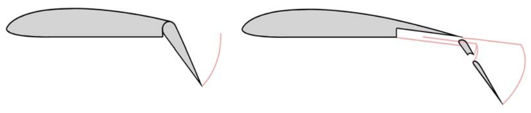
Fig. 3. Plain and double-slotted flap deployment [7]

E. Sensors

Whereas an actuator is an output device that changes state on command, a sensor is its complement as an input device to indicate this state by querying the actuator on command. Sensors do not play a significant role in the current manifestation of this work because the logger discussed in the next section already captures state data for analysis. Nevertheless, they do introduce interesting advanced safety considerations into the flight-control system for future work on model reliability, for example.

For any number of reasons beyond the scope of this paper, a real-world mechanical system may erroneously exceed its specified design limits. An active preventative solution, in the form of a watchdog device, would immediately report any deviation as a fault [8]. A passive solution is also commonly present as mechanical stops that would physically prevent an actuator from exceeding its hard limits. However, it would still be possible to hit a stop and continue to try to move, likely resulting in wear or damage in the drive mechanism. A common hybrid approach is to monitor the actuator itself to see how hard it is working with respect to how much work it is performing: working hard with no effect likely means that it is at the limit, jammed, or otherwise interrupted. A garage-door opener is a good example.

F. *Controllers*

A fly-by-wire system is actually a hierarchical system of systems [9]. In this model, there is one master bus that runs from the cockpit throughout the entire aircraft, but no actuators are connected directly to it. Rather, it consists of controllers, which themselves have slave subbuses containing relevant actuators. This extra level of indirection allows for arbitrarily complex coordination at the receiving end (the actuators), where otherwise it would be the transmitting end (the cockpit) that would have to assume this responsibility. It also reduces bus traffic by sending consolidated requests to be interpreted, decomposed, and redistributed by the controllers as appropriate. The controllers are:

- *Rudder controller*: always contains a single rudder actuator on its subbus. A request to it (i.e., change deflection angle) passes untouched to the actuator.
- *Elevator controller*: always contains two elevator actuators. A request to it passes untouched to each.
- *Gear controller*: always contains two main-gear actuators and a nose-gear actuator. A request to it (extend or retract) passes untouched to each.
- *Flap controller*: contains an even number of symmetrically configured flap actuators, evenly distributed across the wings from left to right from the cockpit perspective. A request to it (downward deflection angle) passes untouched to each.
- *Engine controller*: contains any number of symmetrically configured engine actuators, distributed the same as flaps. A request to it (power percentage) has two interpretations. In gang mode, it passes untouched to each actuator; in isolation mode, it passes untouched to only the specified one.
- *Aileron controller*: contains an even number of symmetrically configured aileron actuators, also distributed the same as flaps. In standard roll mode, a request to it (upward or downward deflection angle) passes untouched to all actuators on the left wing, but the direction is inverted for the right wing. In mixed roll mode, the request addresses only the specified left actuator, and the other actuators on the left wing deflect as a ratio of it, and likewise those on the right. (Section VII discusses this process in more detail.) Finally, in speed-brake mode, the request (deploy or retract) passes to all actuators to deflect upward to their maximum angle or downward to neutral, respectively.

Fig. 4 is an architecture for a typical two-engine passenger airplane, where L, R, and N respectively indicate left, right, and nose. Section V addresses the command generator.

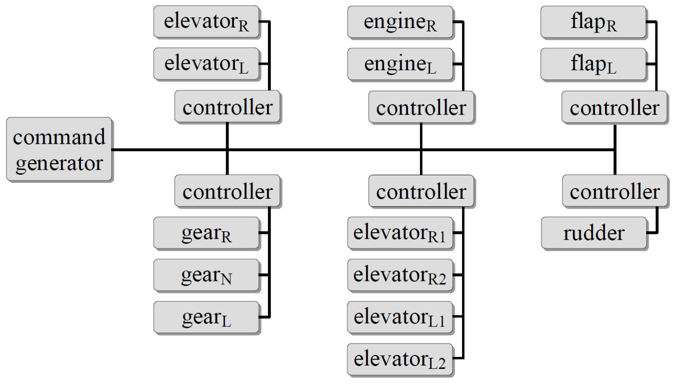

Fig. 4. Typical bus architecture

Additional levels of indirection (i.e., controllers containing controllers) could also be added for complicated decision-making. For example, in flight the appropriate amount of yaw from the rudder depends on the amount of roll from the ailerons with respect to airspeed. An aileron request could issue a secondary rudder request [10]. Similarly, roll reduces lift, which may be compensated for by the pilot or the control system. In general, Boeing's approach, for example, requires the pilot to introduce more pitch as a separate intentional action, whereas Airbus's does so automatically [11].

Although not part of this work, controllers in conjunction with sensors could monitor the behavior of actuators in concert. Symmetric configurations must be in identical (or identically inverted) states at all times; otherwise, the aerodynamic effects could be catastrophic. Similarly, advanced coordination of an autothrottle is possible [12].

Other complex monitoring relationships are also possible, but are deferred to future work. For example, in combination with flight data, the flight-control system could determine the operating limits that the pilot is permitted to reach [4]. Under normal circumstances, operating in so-called Normal Law, the system does not allow the aircraft to enter a region of unusual or diminished flight control. For rare cases, Alternate Law relaxes these restrictions, but it still would not allow the aircraft to exceed its maximum operating limitations. Direct Law basically disables automated oversight altogether for exceptional cases and allows the actuators to be driven as if the control system were purely mechanical.

IV. View

The view module of the model-view-controller architecture depicts the state of the system in multiple forms. Its plug-and-play nature allows other implementations to be added or substituted relatively easily.

A. *Log*

The most basic — but also most informative — view is text output to a log file. This view supports arbitrary logging of any aspects of interest, but the built-in output is generally sufficient for most analysis. The relevant details of each state at each clock tick go to a text file that directly exports to Excel. Fig. 5 shows an abridged form, which actually contains many more columns, as well as typically thousands of rows of events.

tick	time	code	action	bus	servicer_id	s#	request	request_id	status	response	t#
53	0.053	C	submit	bus1	gear_ctrl1	0	Service	gear_ctrl1#1	UNBOUND		
53	0.053	C	submit	gear_ctrl1_bus	gear_nose2	0	Service	gear_nose2#2	UNBOUND		
53	0.053	D	respond	gear_ctrl1_bus	gear_nose2	0	Service	gear_nose2#2		ACCEPTED_SERVICING	
53	0.053	C	submit	gear_ctrl1_bus	gear_main1	0	Service	gear_main1#3	UNBOUND		
53	0.053	D	respond	gear_ctrl1_bus	gear_main1	0	Service	gear_main1#3		ACCEPTED_SERVICING	
53	0.053	C	submit	gear_ctrl1_bus	gear_main2	0	Service	gear_main2#4	UNBOUND		
53	0.053	D	respond	gear_ctrl1_bus	gear_main2	0	Service	gear_main2#4		ACCEPTED_SERVICING	
53	0.053	D	respond	bus1	gear_ctrl1	0	Cancel	gear_ctrl1#1		ACCEPTED_SERVICING	
53	0.053	B	notify	gear_ctrl1_bus	gear_main1	1	Service	gear_main1#3	SERVICE		
53	0.053	B	service	gear_ctrl1_bus	gear_main1	1	Service	gear_main1#3	BLOCK		0
53	0.053	B	notify	gear_ctrl1_bus	gear_nose2	1	Service	gear_nose2#2	BLOCK		
53	0.053	B	service	gear_ctrl1_bus	gear_nose2	1	Service	gear_nose2#2	SERVICE		1
53	0.053	B	notify	gear_ctrl1_bus	gear_main2	1	Service	gear_main2#4	SERVICE		
53	0.053	B	service	gear_ctrl1_bus	gear_main2	1	Cancel	gear_main2#4	SERVICE		2
53	0.053	B	notify	bus1	gear_ctrl1	1	Service	gear_ctrl1#1	CANCEL		
54	0.054	B	notify	gear_ctrl1_bus	gear_main1	2	Service	gear_main1#3	SERVICE		
54	0.054	B	service	gear_ctrl1_bus	gear_main1	2	Service	gear_main1#3	SERVICE		3
54	0.054	B	notify	gear_ctrl1_bus	gear_nose2	2	Service	gear_nose2#2	SERVICE		
54	0.054	B	service	gear_ctrl1_bus	gear_nose2	2	Service	gear_nose2#2	SERVICE		4
54	0.054	B	notify	gear_ctrl1_bus	gear_main2	2	Service	gear_main2#4	SERVICE		
54	0.054	B	service	gear_ctrl1_bus	gear_main2	2	Service	gear_main2#4	SERVICE		5
54	0.054	B	notify	bus1	gear_ctrl2	2	Terminate	gear_ctrl1#2	TERMINATE		

Fig. 5. Abridged bus log

Not only does this log represent the states of the controllers and actuators, but it also shows the bus traffic. The communication process involves submitting a request and getting one or more immediate responses from its servicer. If the request is subsequently serviced, notifications are sent back to the requester for specified conditions like decelerating for arrival at the final state, arrival at the final state, preemption, and so on. This information could be exported to other applications to generate timing and UML sequence diagrams.

B. *Graph Visualization*

The textual form of the log file is richly informative, but it is not intuitively understandable. However, its structure not only exports natively to Excel for a tabular representation; its fields are also strategically organized to allow event chains to be plotted as line graphs. Fig. 6, for example, depicts the actions of a rudder actuator at the following key time points:

1. at initial position 0º neutral; command to 45º left
2. arrives; command to 45º right
3. arrives; command to 0º
4. arrives; command to 30º left
5. at 15º left preemptively command to 45º right
6. arrives

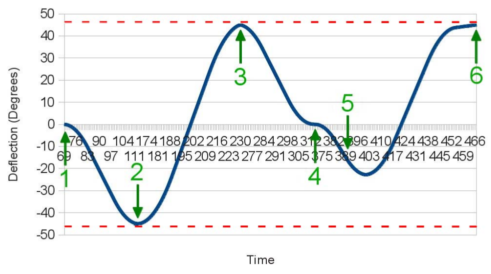

Fig. 6. Preemption test

Despite multiple requests, it is clear that at no time does the actuator find itself in an inconsistent state. i.e., above or below the dashed lines depicting the physical limits. Moreover, the line represents a smooth, continuous function with no jerks or breaks through various acceleration, deceleration, and preemption actions. Such intuitive visual inspection is invaluable for testing and evaluation. Furthermore, mathematical analysis on the slope of the function would demonstrate that the performance remained within the specifications at all times.

C. *Three-Dimensional Visualization*

Visual representation in graph form is useful with respect to a localized part of the system like a single actuator or a group of related actuators. However, for a global systems-level view, three-dimensional visualization, as in Fig. 7, is far better.

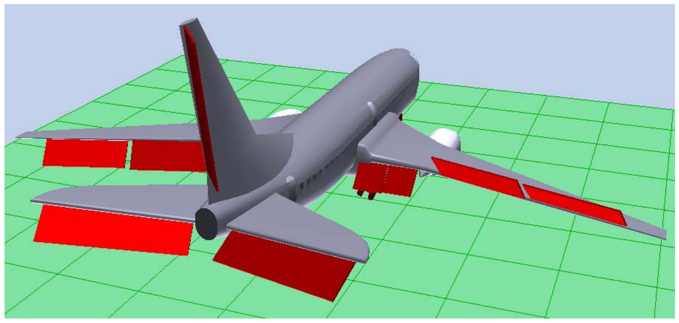

Fig. 7. Actuator visualization [1]

As the next section discusses, actuator configuration is highly dynamic and need not correspond to any particular aircraft. As a result, the visualizer is stylized with oversize control surfaces that represent the appropriate actions, not necessarily the appropriate appearance. The visualizer can also depict metadata like physical limits and breadcrumb tracks showing a history of state changes.

This OpenGL-based Java tool has seen extensive use in the author's artificial-intelligence and software-engineering courses, related pedagogical research, and industry work as a general-purpose world viewer [13,14]. It is freely available at shelby.ewu.edu.

V. CONTROLLER

The controller is the user interface for building the model and running the simulation. It is based on a regular grammar that employs the Interpreter and Command design patterns [2]. The instructor's solution defines the parser with JavaCC, but the students had to design and implement their own with standard Java. This effort entailed thoroughly understanding the problem domain of the commands and their proper usage, as well as the solution domain of the API for the provided architecture. It strongly discourages head-first, brute-force coding by making such an undisciplined approach obvious and unpleasantly difficult.

A. *Creational Commands*

Creational commands define and build the actuators via the Builder and Factory design patterns [2]. Each contains a unique identifier, the interval limits, a delta value for changing state on the interval, and an acceleration for changing the delta value.

All eight of these commands (for aileron, elevator, and rudder actuators, etc.) have the following form:

```
CREATE RUDDER id WITH LIMIT angle
  SPEED speed ACCELERATION acceleration
```

B. *Structural Commands*

Structural commands define and build the controllers from the actuators created above. Each contains a unique identifier and the actuators. The controllers with a fixed number of actuators are:

```
DECLARE RUDDER CONTROLLER id1
 WITH RUDDER id2

DECLARE ELEVATOR CONTROLLER id1
 WITH ELEVATORS id2 id3

DECLARE GEAR CONTROLLER id1
 WITH GEAR NOSE id2 MAIN id3 id4
```

The controllers with a variable number of actuators are:

```
DECLARE FLAP CONTROLLER id WITH FLAPS idn+

DECLARE ENGINE CONTROLLER id
 WITH ENGINE[S] idn+

DECLARE AILERON CONTROLLER id
 WITH AILERONS idn+ PRIMARY idx
 (SLAVE idslave TO idmaster BY
 percent PERCENT)*
```

All controllers must be added to the master bus:

```
DECLARE BUS id WITH CONTROLLER[S] idn+
```

And finally the configuration is locked to prohibit further creational or structural commands and to authorize most behavior commands:

```
COMMIT
```

It is at this point that students perform any late consistency checks to verify that the system is configured properly before any manipulation of it can occur. For example, the number of engines and their properties must mirror each other on the wings. From a practical design standpoint, such a check cannot be done earlier while the engines are still being added because the process is sequential. Determining what can be done immediately versus deferred for later, as well as how, is an important part of software design thinking [15].

C. *Behavioral Commands*

The behavioral commands send requests across the master bus to be interpreted by the appropriate controller(s):

```
DO id DEFLECT RUDDER angle LEFT | RIGHT
DO id DEFLECT ELEVATOR angle UP | DOWN
DO id DEFLECT AILERONS angle UP | DOWN
DO id SPEED BRAKE ON | OFF
DO id DEFLECT FLAP position
DO id SET POWER power
DO id SET POWER power ENGINE id
DO id GEAR UP | DOWN
HALT id
```

D. *Miscellaneous Commands*

The miscellaneous commands manipulate the execution of the simulation. The first set affects the system clock:

```
@CLOCK rate
@CLOCK PAUSE | RESUME | UPDATE
```

Especially important for testing is the capability to wait a fixed amount of time. The preemption tests, in particular, need to be timed exactly. Manually issuing a command would result in inconsistent results over multiple runs. The command is:

```
@WAIT time
```

Finally, commands may be supplied in text files for execution as scripts, which greatly simplifies repeatable testing:

```
@RUN "filename"
```

VI. Controlled Experiments

The primary goal of this work is to instill disciplined software-development behavior in students. However, it also naturally serves as an effective platform for systematically evaluating the performance of a model by using scientific method in controlled experiments as follows:

1. Design and carry out an experiment (a test) to investigate something of interest.
2. Visualize and analyze the results.
3. If the results are unsatisfactory, perturb one and only one parameter in the experiment and rerun it.
4. If the new results are more promising, continue down this line of investigation; otherwise, either perturb the parameter in a different way or reset it and perturb a different parameter.
5. Continue to refine the model until the results are satisfactory.

Consistent with the primary goal, students gain valuable experience with discovering patterns and building mental models that connect causes to effects [16]. This approach reduces the amount of random, uninformed generate-and-test cycles, which generally allows students to be more productive by getting better results with less effort.

VII. Results

The students' project consisted of three parts: implement the parser, implement the controllers, and execute a detailed test plan. The results of the first two parts were straightforward because this system does not allow them to deviate much from proper coding principles. The third part — testing and evaluation of the entire system — is the emphasis here. The deliverable required a formal report describing the test plan and its results. Each of 27 experiments addressed eight points, where 1–4 related to planning, 5–6 to execution, and 7–8 to presenting the results:

1. The rationale behind the test; i.e., what it was testing and why it mattered.

2. A general English description of the initial conditions.
3. The commands for (2).
4. An English narrative of the expected results.
5. The actual results with at least one graph showing the most representative view of the states.
6. A snippet of the actual results from the log file with a supporting explanation, including statistics, metrics, and graphs, as appropriate.
7. A discussion on how the actual results agreed with the expected results, or if they disagreed, a hypothesis on why.
8. A suggestion for how to extend this test to address related aspects of potential interest.

The results themselves are not as relevant here as what the students learned from them, but the following examples help convey how they learned it. Fig. 8 depicts the behavior of a three-engine configuration (where the dashed lines overlap). At time point (1), all three engines were commanded to increase power from 0 to 70%. Upon reaching this target at (2), the center engine was commanded to reduce power back to 0%. At (3), while the center engine was still decreasing, all three were commanded to go to 100%, which took until (4) to achieve.

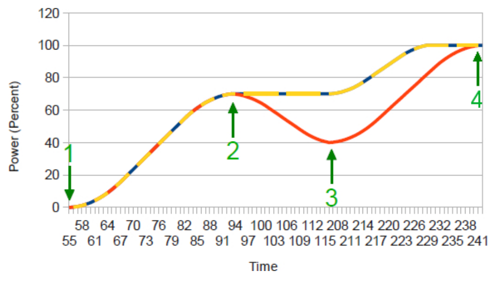

Fig. 8. Engine manipulation

Determining correctness (testing) and evaluating performance (optimization) require three critical components: the expected results, the actual results, and a meaningful way to compare the two. All too often students lack one, two, or even all three of these. While the technical act of acquiring such data is necessary, it alone is not sufficient to make sense of the data. Students must have a firm understanding of the subject matter and its context within the problem domain. The pedagogical approach in this work provides endless opportunities to ground the programmatic exercises to reality in order to help students develop and improve their critical-thinking skills in computer science.

For example, experiments in engine manipulation could be better understood by knowing that it is normal procedure for pilots of many commercial airliners to spool the engines initially to 50% power on takeoff and then wait until they all reach this point before going to full power. The natural variability in engine performance does not guarantee exactly the same behavior simultaneously, which could have adverse effects on controllability during the takeoff roll. With this knowledge, students may realize that they are not just generating graphs; they are generating graphs that mean something, and if that something is not what it should be, then they have considerable insight into what may need fixing.

Another example in Fig. 9 shows a complex example of mixing eight ailerons with different performance properties in roll mode. At (1), with all ailerons at their neutral position of 0 degrees, the master aileron (M) is commanded to deflect upward to 45 degrees. The slave ailerons on the same wing move upward as a ratio of its movement, while those on the opposite wing correspondingly do so downward. Upon reaching 45 degrees at (2), the master is commanded downward to –40 degrees, which is achieved at (3). The graph convincingly shows that all ailerons remain antisymmetrically synchronized at all times.

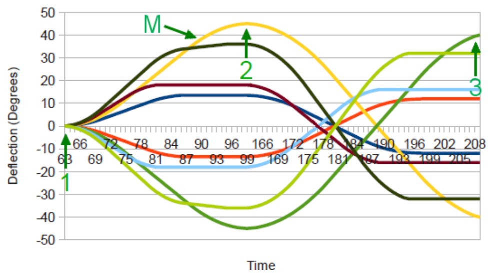

Fig. 9. Aileron manipulation

Fig. 10 shows the same ailerons acting in speed-brake mode. Starting again from neutral 0 degrees at (1), the eight ailerons deflect upward to their maximum limit of 90 degrees, which is achieved by all by (2). The master is then commanded to –20 degrees in roll mode, which results in behavior analogous to that in Fig. 9. The details of the action are beyond the scope of this paper, but again, the graph clearly demonstrates them to the students, who have the theoretical and practical foundation to know how to interpret it.

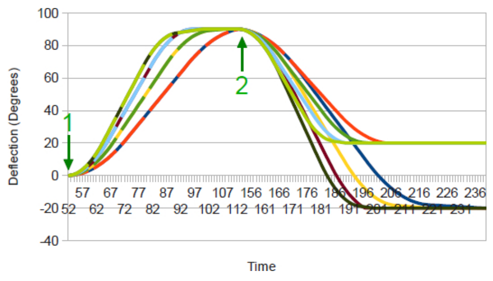

Fig. 10. Speed-brake manipulation

Finally, Fig. 11 shows a four-flap configuration, where two types of flaps differ in performance slightly. At all times through a variety of actions, the flaps of the same type remain synchronized, and the two types act appropriately with respect to one another. Such insight into the behavior of a complex system is invaluable in testing and evaluation.

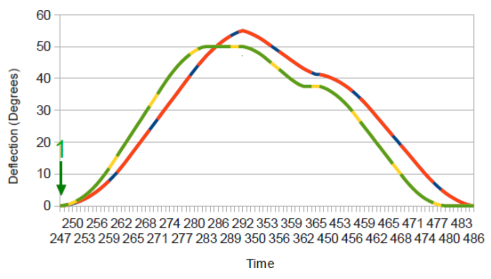

Fig. 11. Flap manipulation

Reporting and evaluating the results of human subjects in work of this scope must be summarized due to limited space. However, it is based on a significant breadth and depth of objective and subjective measures including anecdotal observation, individual contributions from a background survey, 11 assignments, and 10 anonymous weekly assessments, as well as individual and team contributions from 18 project status reports, a project reflection, a team evaluation, a self-evaluation, and a course evaluation.

Most telling, 90% (28 of 31) of the participating students indicated that the graphical form of the test report directly contributed to a better understanding of what their code was actually doing, where they otherwise would have had less confidence in their results. Overall, the students rated the project 4.6 out of 5 (excellent).

VIII. Future Work

Almost any complex physical system is characterized by input, processing, and output that could map to this architecture. For reuse in future projects, it should accommodate more complex interval behaviors, as well as other sensors, actuators, and controllers. Of particular interest within this application are detecting and handling faults and managing operating laws. For a graduate-level course, especially for software quality assurance, the evaluation framework based on controlled experiments could be significantly expanded for richer analysis. In particular, the introduction of probability would contribute to a powerful stochastic Monte Carlo methodology.

IX. Conclusion

This work mitigates the common problem-solving strategy of many students who undermine the design of a system by indiscriminately throwing more code at every issue they encounter. The hierarchical quasi-network-based architecture effectively prevents them from communicating with their components by any means except the prescribed ones. Its datatype-oriented interval framework safely and elegantly manages complex physical behaviors concurrently. This unified approach, combined with an overarching framework of modeling, simulation, visualization, and analysis, further provides a disciplined strategy for creating, executing, presenting, and analyzing meaningful test cases as part of formal test plans based on a sound methodology of controlled experiments via scientific method. Extensive results from a classroom deployment support the conclusion that students overwhelmingly benefit from this approach.

References

[1] Adapted from Google Sketchup Warehouse, 3dwarehouse.sketchup.com, last accessed 21 Apr. 2015.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Indianapolis, IN, 1995.

[3] C. Okasaki. Purely Functional Data Structures. Cambridge University Press, New York, 1999.

[4] W. Phillips. Mechanics of Flight. Hoboken: Wiley, 2009.

[5] S. McConnell. Code Complete: A Practical Handbook of Software Construction. Microsoft, Redmond, 2004.

[6] M. Cook. Flight Dynamics Principles. Elsevier, Waltham, MA, 2013.

[7] Adapted from Wikipedia, wikipedia.org/wiki/Flap_(aircraft), last accessed 21 Apr. 2015.

[8] R. Isermann, R. Schwartz, and S. Stolzl. “Fault-tolerant drive-by-wire systems.” IEEE Control Systems 22(5), 2002.

[9] S. Pope. “Fly by wire: Fact versus science fiction.” Flying, pp. 53–59, May 2014.

[10] M. Napolitano. Aircraft Dynamics: From Modeling to Simulation. Hoboken: Wiley, 2011.

[11] R. Collinson. Introduction to Avionics Systems. Springer, Netherlands, 2011.

[12] S. Pope. “Autothrottle advances.” Flying, pp. 66–69, April 2015.

[13] D. Tappan. “A pedagogical framework for modeling and simulating intelligent agents and control systems.” Technical Report WS-08-02, AAAI Press, 2008.

[14] D. Tappan. “Student-friendly Java-based multiagent event handling.” In Proc. of Association for the Advancement of Artificial Intelligence, Bellevue, WA, 2013.

[15] A. Grosskopf, M. Weske, J. Edelman, M. Steinert, and L. Leifer. “Design Thinking implemented in software engineering tools.” In Proc. of 8th Design Thinking Research Symposium, Sydney, Australia, 2010.

[16] M. Salmon. Introduction to Logic and Critical Thinking. Cengage Learning, Boston, MA, 2013.