Paper Presented at AlaSim 2023

A Mechatronics Virtual Testbed for Investigating Concepts and Practices in Software Engineering Education

Dan Tappan
Department of Computer Science, Eastern Washington University, Spokane, WA
Keywords: software engineering, mechatronics, pedagogy

ABSTRACT: *Modern engineering problems are addressed by highly multidisciplinary solutions involving mechanical and electrical engineering with software engineering at the core, a synergistic combination known as mechatronics. Computer science and software engineering students do not generally receive exposure to the holistic process of developing complex software to drive such hardware systems. This work provides a virtual testbed environment for modeling and simulating an extensive breadth and depth of common concepts and practices. It allows them to design, build, manipulate, visualize, and analyze arbitrary mechatronic systems from a software engineering perspective. The primary example investigates its usage in developing a fly-by-wire aircraft control system.*

1 Introduction

Software is the heart of modern multidisciplinary engineering systems. Computer scientists and software engineers therefore play a critical role in developing them. However, as students they receive very little theoretical and practical experience in understanding the problems such systems address and their corresponding solutions.

The objectives of this work are to provide a student-friendly modeling and simulation toolkit that allows them to build realistic virtual systems and to analyze them before, during, and after development. It allows students to play the roles of analyst for design, programmer for implementation, and end user for test and evaluation.

2 Problem Domain

The breadth and depth of real-world problems this system can address is vast, but they all fall under the umbrella of mechatronics. This field is relatively new, at least as an established term, but it reflects what engineers have always done to solve multidisciplinary problems [9]. Specifically, it refers here to the bridge in Figure 1 between computer science (CS) and the real world. CS does not interact directly with the world (although a good system appears this way to the end user). Rather, it interacts with the electrical engineering (EE) layer, which in turn interacts with the mechanical engineering (ME) layer, which finally interacts with the world.

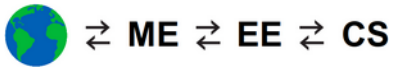

Figure 1: Mechatronics

Throughout the historical evolution of engineering solutions, each new layer has appeared to the right and taken on many of the responsibilities of those to the left. For example, purely mechanical control systems were managing steam engines well before electrical systems appeared. The physical ME and EE layers have always been limited in what they can provide given their hardware constraints, whereas the virtual CS layer is for all practical purposes unlimited in software. This evolution explains, for example, why today ordinary automobiles are heading toward 300+ million lines of programming code managing almost every aspect of operation [2].

With the exception of minor excursions into shared EE topics like digital design, CS students study exclusively CS. They are not expected to be experts in the other solution layers or the problem layer from the real world. However, they personally will have to interact with those practitioners in their careers, and their solutions will have to interact with those solutions. Such is the nature of multidisciplinary work. It is critical for students to understand that all four layers address the exact same problem from four different perspectives. To function effectively as a system – of both people and technology – they must have exposure to such problems and a healthy appreciation for learning outside the comfort zone of their formal education. This system is designed specifically to address these aspects.

2.1 Theoretical Foundation

The theoretical foundation briefly covers directly relevant aspects of how to learn effective problem solving in software engineering, as defined above.

Anecdotally, based on 10 years of industry experience and nearly 20 in academia, the author considers the two top problems in software engineering to be:

#1 Not understanding the customer's problem
#2 Not understanding the customer's problem domain

A large part of this lack of understanding is that students do not really know what the science in computer science is or the engineering in software engineering. Despite taking required science classes, most students are of the opinion that the class is only about the subject matter, for example, rocks in geology, and not about learning how scientists think and act. Students even have this misconception about computer science itself. Edsger Dijkstra, one of the fathers of computing, sums it up well: "Computer science is no more about computers than astronomy is about telescopes" [10].

It is critical to connect this background properly because it plays an important role in modeling and simulation. In particular, this work considers the following definitions, where each is the study of:

Science: how existing natural systems work
Engineering: how to create new artificial systems

In general, science plays the role of analysis, and engineering plays the role of synthesis. Both involve a well-defined process to work from the question or problem to a corresponding solution. In science, this process is called the scientific method, as depicted in Figure 2.

Figure 2: The Scientific Method [11]

The complementary process, the engineering method in Figure 3, is almost identical. Both are also the basis for how modeling and simulation works. Students see that all three processes are effectively the same and already familiar. This system helps them identify suggested actions at each stage with respect to the context of the problem.

Figure 3: The Engineering Method [11]

Much like solving algebraic problems, both methods contain elements that practitioners have (from the problem), want to have (as the solution), and need to get there, as well as a way to obtain what is missing. General categories for this process are based on what is known and on what is known about it, called metacognition [4]:

Things...	**we know**	**we do not know**
we know	facts	questions
we do not know	intuition	exploration

Table 1: Metacognition

Read by row then column, Table 1 captures four possibilities. For each, there are different ways to obtain the desired results in increasing level of difficulty; for example:

Facts: ask Google or ChatGPT
Questions: ask the customer; use modeling/simulation
Intuition: rely on experience
Exploration: use modeling/simulation

Two cases use modeling and simulation. Questions are generally lighter weight, where a proof of concept or by construction may produce a useful answer. Exploration is more complex, often involving analysis of alternative approaches in trade studies, for example.

Intuition relies on experience, which is precisely what students do not have. Facts, which should involve a straightforward process of looking up the answer, are oddly problematic because students often feel background research into the problem and its domain are nothing but busy work.

To support metacognition, there are also familiar linguistic features in the form of the W^5H question words *who*, *what*, *when*, *where*, *why*, and *how* that help students

pose the right questions the right way at the right time to extract the right details. This process is actually very difficult in general (e.g., eliciting requirements), and especially among predominantly introverted CS students. The more practice they get at technical communication in various forms, the better.

The final theoretical element is the DIKW hierarchy of *data*, *information*, *knowledge*, and *wisdom* in Figure 4. It corresponds closely to everything above as a stepping-stone approach to understanding the problem and its domain by establishing and connecting dots. It slices and dices the subject matter from different perspectives from the lowest to the highest levels compositionally. This system considers data as isolated facts, information as the association of a few facts in a localized context, knowledge as the association of multiple contexts in the bigger picture, and wisdom as an understanding of the entire system of contexts.

Figure 4: DIKW Dots

The more common visual form is the DIKW Pyramid or Hierarchy in Figure 5. It aligns with the dot form as context on the left slope. This composition applies both to the process of learning about the problem itself, as well as to the process of solving it, as the next section describes with design patterns.

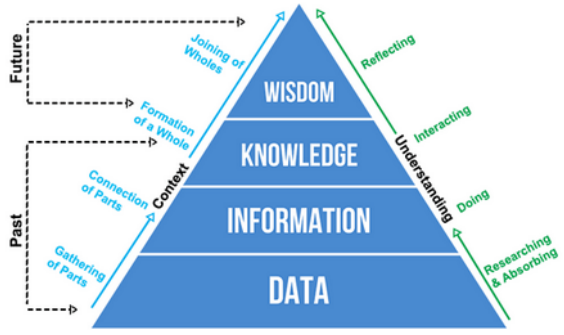

Figure 5: DIKW Pyramid 1 [8]

The right slope aligns with actions to be taken at each level. It corresponds with metacognition and the problem-solving methods discussed above. The aspects of past and future refer to the learning process, where things are unfamiliar and often overwhelming the first time they are encountered, but they are absorbed into wisdom and expertise over time.

The alternative representation in Figure 6 similarly captures understanding on the left slope, but it also elaborates with further linguistic aspects on the right to show generally what students should be looking for or working with at each level. Finally, this figure shows the continuous reduction of overall risk as the understanding of the problem and its domain increases. As mentioned earlier, students have an odd aversion to the lower levels, which are unfortunately where the risk is the highest. The pedagogical approach in this work forces them to perform at each level.

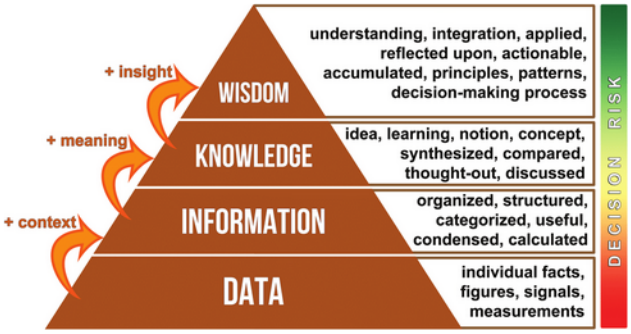

Figure 6: DIKW Pyramid 2 [3]

2.2 Practical Foundation

The practical foundation briefly covers directly relevant aspects of how to apply effective problem solving in software engineering. The primary framework is called QMSVA, which reflects the scientific and engineering methods, as well as traditional modeling and simulation:

Question:	the question or problem to address
Modeling:	a representation of the problem
Simulation:	execution of the model
Visualization:	the output of the simulation
Analysis:	making sense of the output

Every computer program is inherently a model; therefore, students already have extensive experience in creating and executing models without even realizing it. This system emphasizes the connection.

Visualization is actually any form of output, but graphical representations tend to be the most informative and useful. Students can often apply common sense to whether something looks right or wrong[1] without being subject-matter experts or needing extensive background research to understand the problem domain.

Analysis forces students to apply tools and techniques from other coursework, such as critical thinking and math, to make sense of the results with respect to the problem or question.

The secondary framework is closely based on the concept of software design patterns [5]. There are dozens of such "mini-solutions" for commonly encountered design

[1] Sometimes called the TLAR approach: *that looks about right*.

considerations. The details are not relevant here, though. What is important is their organization into three hierarchical categories:

Creational patterns:	making components
Structural patterns:	connecting components
Behavioral patterns:	using connected components

Components are basically the dots in Figure 4. Once students learn to recognize (from the science perspective) the composition of the problem domain, it becomes much easier to apply (from the engineering perspective) the appropriate solutions in the solution domain. In fact, entire books are dedicated to this way of thinking, as in Figure 7.

Figure 7: Everyday Engineering [6]

Conversely, not having the right dots and/or not connecting them properly leads to incorrect behavior. The software industry has an atrocious failure rate between 50 and 90%, depending on the size of the project [7]. Many of the errors in thinking and doing by professionals develop when they are students. This system attempts to mitigate many of the causes.

3 Solution Domain

The solution domain is a Java program that provides all the capabilities available to the instructor and students to address the problem domain. The breadth and depth of real-world engineering systems it can address is virtually unlimited. However, despite this range, every supported system is based on the same simple concept in Figure 8, which captures bidirectional motion between two points.

Figure 8: Conceptual Movement

In more concrete terms, A and B define a number line, on which the orange dot resides. This simple representation can then map onto a vast array of mechanical mechanisms and devices, as showcased in the book in Figure 9, for example.

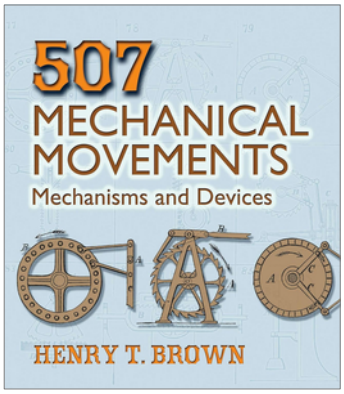

Figure 9: Mechanical Systems [1]

Students can easily see that the mechanical engineering layer is not actually as foreign and scary as they initially believe. Furthermore, this approach demonstrates that one solution can be applied to a large number of seemingly disparate problems. This principle is highly valued in designs, but from inexperience, students naturally tend to address each problem with a unique solution. The result is usually bloated software that is difficult to design, implement, test, and maintain.

3.1 MVC Architecture

The system is based on the model-view-controller (MVC) architecture in Figure 10. It is the basis of many, if not most, modern software solutions. The model plays the same role as in modeling and simulation, namely the thing being represented; it is also the M in QMSVA. The view is some representation of the output from the model, and the V. The controller is a mechanism to manipulate the model, and the S.

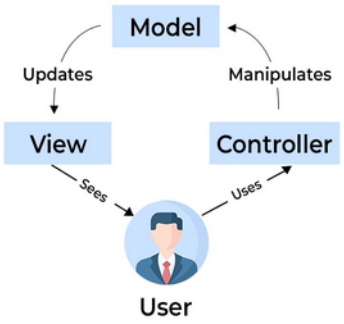

Figure 10: MVC Architecture [11]

Again, this single solution accommodates three variants of problems. Figure 10 shows the human in the loop to operate the controller. This role can also be a “dumb” automated component, like an ordinary thermostat, or a “smart” autonomous component, like an AI entity. The variants allow the instructor to use this system, and in fact, often even the same examples, in multiple courses from different perspectives for different purposes.

3.1.1 MVC Model

The MVC model is a hierarchical network of conceptual components in Figure 8 mapped onto physical mechanisms, such as those in Figure 9. The running example here is the fly-by-wire aircraft flight control system in Figure 11. The mechanisms include the labeled control surfaces, as well as the landing gear and engines.

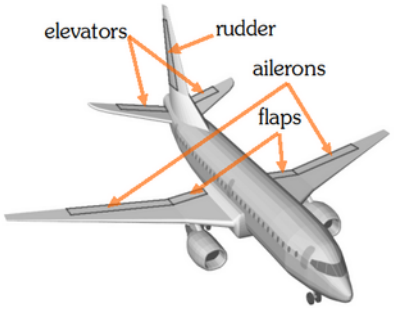

Figure 11: Simulated Airplane

Each of the boxes in Figure 12 is a component or collection of supporting components that the next sections describe.

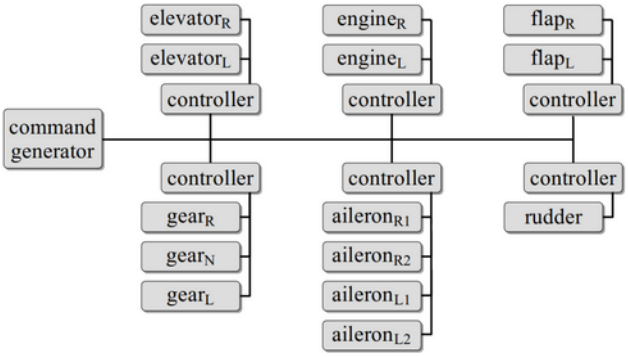

Figure 12: Hierarchical Network

Boxes are the creational elements, and the network is the structural element connecting them. The behavioral element is making use of the system as constructed.

3.1.1.1 Primary Components

Primary components are required because they define the minimum operational capabilities.

3.1.1.1.1 Actuator

All movement of any kind is effected by actuators. These can be considered as motors that provide the capability to move between points A and B. Figure 13 shows a single actuator and a snippet of the network. The required controller manages one or more actuators because actuators are inherently dumb devices that make no decisions for themselves. The next section covers the controller in more detail.

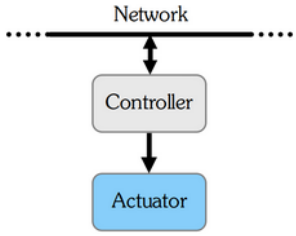

Figure 13: Actuator Connection

Figure 14 shows the actuators in blue that correspond to Figure 11.

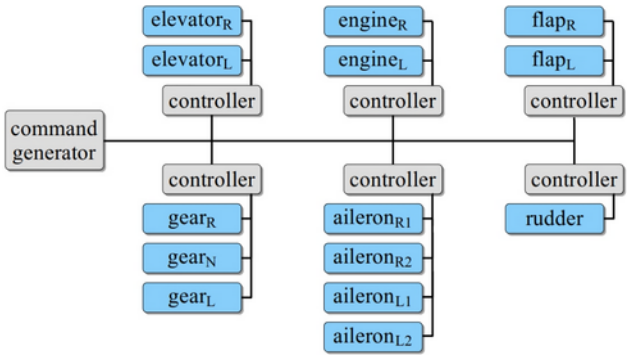

Figure 14: Actuator Network

Figure 8 describes linear movement between points A and B. There is also the second variant available in Figure 15 for rotary movement. It plays the same role, but without the restriction of end points. This configuration captures the movement of the engines, for example.

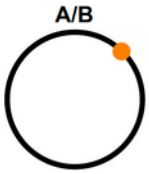

Figure 15: Rotary Movement

Either form of movement can be configured to reflect real-world performance. Figure 16 shows a variety of examples. The *x*-axis is time and the *y*-axis is based on the actuator configuration. The blue lines are position; the red are speed.

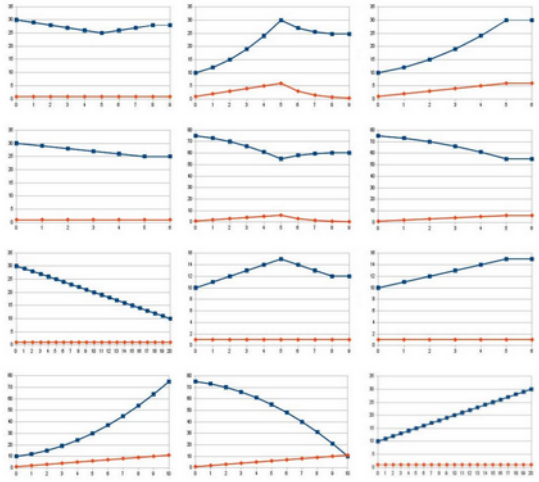

Figure 16: Movement Configuration

3.1.1.1.2 Controller

Each controller[2] in Figure 13 and Figure 14 manages one or more actuators. This organization allows related components to have localized authority over well-defined subsystems. For example, Figure 17 has a controller for all three parts of the landing gear (left, right, and nose). A command to lower the gear collectively (the *what* action) goes to the controller, which then determines the details of individual actuator movements (the *how* action). Controllers can also manage subcontrollers for advanced systems.

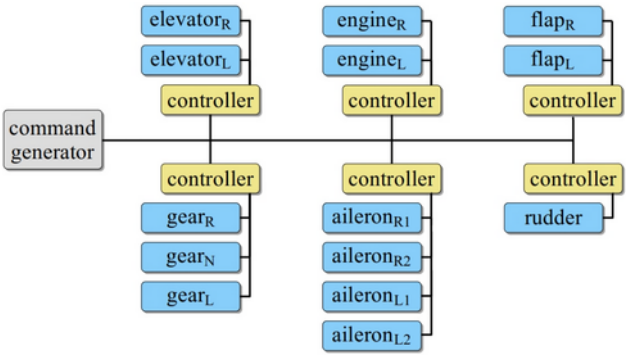

Figure 17: Controller Connection

3.1.1.1.3 Message

The command to lower the landing gear is a message transmitted over the network, as in Figure 18. The virtual network is very similar to a standard computer network with unique identifiers (like IP addresses) for each component.

[2] This controller is not the same as the MVC controller in Section 3.1.

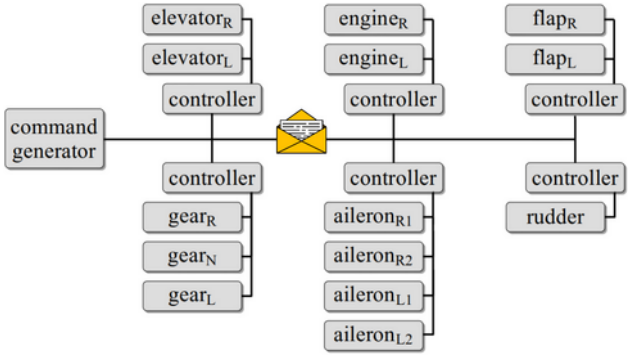

Figure 18: Message Role

Although the conventional term is *command*, most communication is actually in the form of a *request*. This distinction reflects the possibility that a controller may not be able to service a request at the time or even at all, depending on the state of the components it is managing. For example, lowering the landing gear takes a certain amount of time. If a subsequent request is received to raise the gear back up during this process, it cannot be serviced until the lowering is complete. In other cases, a subsequent request may interrupt and cancel the ongoing one and be serviced immediately. Other variants are possible, too.

3.1.1.2 Secondary Components

Secondary components are optional to provide advanced capabilities. They connect to other components and cannot stand alone.

3.1.1.2.1 Sensor

An actuator is a dumb device that does not know its own state. Sometimes this limitation is appropriate, such as with a fan, which does not need to know its current speed or orientation. This open-loop control system assumes the state of an actuator is always as desired, but it has no way to make this determination or to react if it is not true.

In other cases, a closed-loop control system is more appropriate. One or more sensors can be attached to an actuator, as in Figure 19. The sensor reads the state data from the actuator and provides it to the controller, which then decides on the suitable action. For example, the controller for an automobile cruise control verifies that the actual speed is the same as the expected speed. If not, it instructs the engine actuator to speed up or slow down.

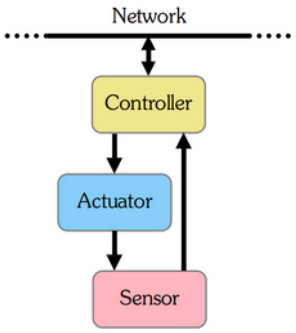

Figure 19: Sensor Connection

3.1.1.2.2 Mapper

A sensor reads the raw state of an actuator. This form may not be directly usable for subsequent processing in the controller. For example, the wheel sensor of the cruise control may measure revolutions per second, but the controller expects miles per hour. Such algebraic manipulation is the role of a mapper. One or more may be connected to a sensor, as in Figure 20.

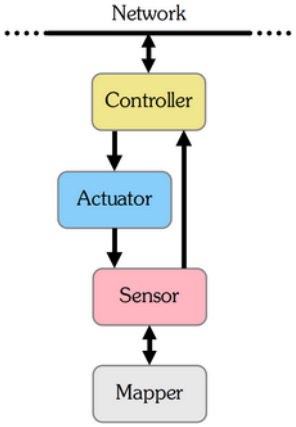

Figure 20: Mapper Connection

In addition to a variety of options for equation-based conversions, it is possible to reference tables in external files that define specialized performance. For example, Figure 21 shows the performance curves for horsepower and torque of an internal combustion engine.

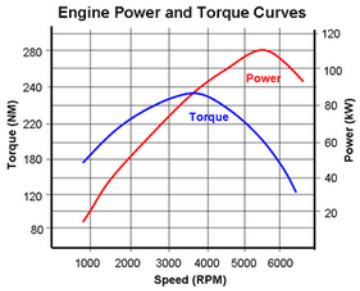

Figure 21: Performance Curves [11]

3.1.1.2.3 Reporter

In order to evaluate the performance of a system, there needs to be a way to export the data generated from its behavior to external files. One or more reporters play this role by connecting to a sensor, as in Figure 22. Section 3.1.3 covers the log files.

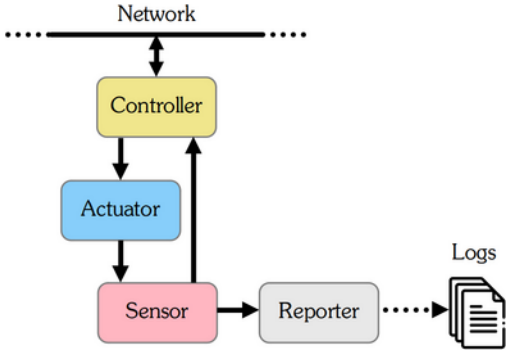

Figure 22: Reporter Connection

3.1.1.2.4 Watchdog

The closed-loop controller in Section 3.1.1.2.1 manages behavior under normal operating circumstances. The role of one or more watchdogs connected to a sensor is to monitor for abnormal behavior, as in Figure 23. If performance is outside a defined range, it raises an alarm. A variety of configuration settings define minimum and maximum values, acceptable changes over time, etc.

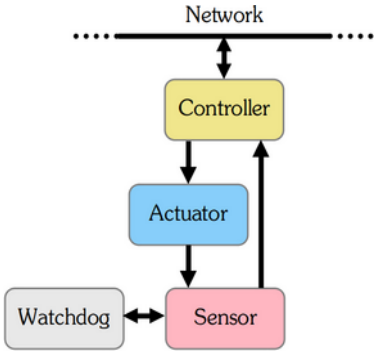

Figure 23: Watchdog Connection

3.1.1.2.5 Summary

Figure 24 shows a complete complement of each primary and secondary component. There is no limit to the number of each in a system, but student projects are generally kept to a reasonably simple level of complexity.

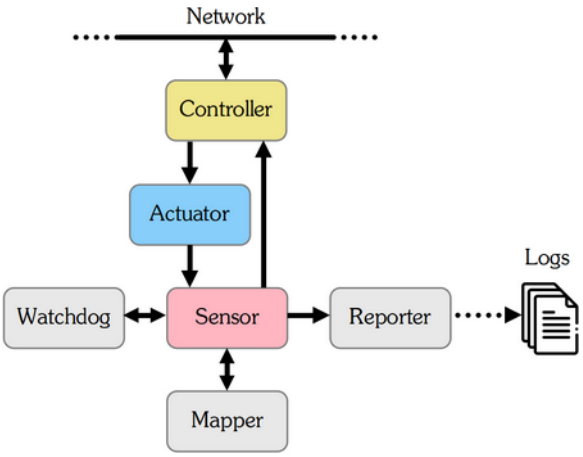

Figure 24: Connection Summary

3.1.2 MVC Controller

The controller in the MVC architecture is where the user interacts with the system to build and execute the model. The command generator in Figure 25 is the top level of the hierarchical network.

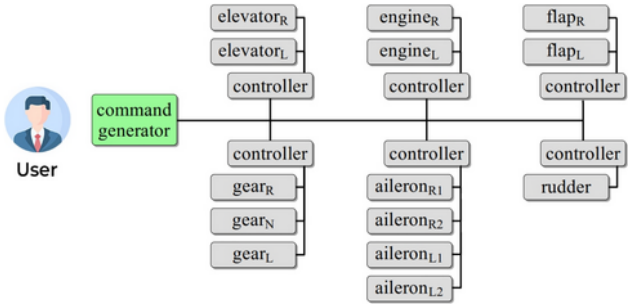

Figure 25: MVC Controller

3.1.2.1 Parser Commands

The user performs all interaction with the system by issuing text commands through a basic command-line interface. There are commands in many variations for each of the categories in Section 2.2. The following sections provide an overview.

3.1.2.1.1 Creational Commands

Creational commands are responsible for defining the components to be added to the network. There are 16. Most have optional arguments to allow for fine tuning, if desired.

For example, the command to create a watchdog with a threshold value looks like this:

```
CREATE WATCHDOG (LOW | HIGH) id mode THRESHOLD
 value₁ [GRACE value₂]
```

Parentheses mean to choose one from the options delimited by a vertical bar. Square brackets mean optional. Lowercase arguments require values or reference additional rules.

The description is: Creates a watchdog with identifier `id` that monitors a value. For the `LOW` variant, the watchdog triggers if the value is less than `value₁`. For `HIGH`, the watchdog triggers if the value is greater than `value₁`. The grace argument states that a violation must be present for `value₂` consecutive clock ticks before raising an alarm. Omitting it reports the first violation.

The `mode` argument defines how to measure `value₁`:

```
MODE (INSTANTANEOUS | (AVERAGE [value₁]) |
 (STANDARD DEVIATION [value₂]))
```

- `INSTANTANEOUS` uses the current value.
- `AVERAGE` uses the average of the last `value₁` values or all values if `value₁` is omitted.
- `STANDARD DEVIATION` uses the standard deviation of the last `value₂` values or all values if `value₂` is omitted.

This functionality for basic real-time statistical analysis offers students a range of options in determining what to measure and how to measure it.

3.1.2.1.2 Structural Commands

Structural commands are responsible for connecting the created components to each other and to the network. There are two forms. One connects secondary components when creating a primary component; e.g.,

```
CREATE ACTUATOR LINEAR a1 SENSORS s1 s2 s3
```

It creates a linear actuator called `a1` with sensors `s1`, `s2`, and `s3` (which must already exist).

The other form performs the connections at the end; e.g.,

```
BUILD NETWORK WITH COMPONENTS c1 c2
```

It adds controllers `c1` and `c2` to the network (along with the components they manage and anything connected to them).

The difference in forms is primarily due to automated error checking of the design. Some checks can be done earlier, while others must wait until everything is available. Examples include duplicate identifiers and components that are unused or used more than once.

3.1.2.1.3 Behavioral Commands

Behavioral commands are responsible for manipulating the fully constructed network. Most relate to messaging. For example,

```
SEND MESSAGE [ids] [groups] POSITION value
```

sends a request to any controllers in the list of `ids` and/or any in the list of `groups` (an arbitrary association of related controllers) to do whatever is appropriate to achieve the state `value`. If neither list is present, the message goes to all controllers. Those that cannot process it ignore it.

The complementary command is to ask a controller or controllers to report their state, which is based on the state of the components they are managing:

```
SEND MESSAGE [ids] [groups] POSITION REPORT
```

There are also several commands that "cheat" by directly forcing components to assume a state or report a value instead of going through the appropriate networking process. For example,

```
SET SENSOR s1 VALUE 9
```

forces sensor `s1` to report output 9 regardless of the actual value from the actuator it is monitoring. This capability allows students to inject errors into the system for advanced analysis.

3.1.2.1.4 Miscellaneous Commands

Miscellaneous commands are responsible for controlling aspects of the system itself. They include, for example, pausing or one-stepping the clock or changing its update rate, executing script files, or exiting the system:

```
@CLOCK PAUSE
@CLOCK ONESTEP 3
@CLOCK SET RATE 20
@RUN "test1.mvt"
@EXIT
```

3.1.3 MVC View

The view in the MVC architecture is where the various forms of output appear. The system generates some views automatically or upon request; others come from reporters selectively connected to components of interest.

3.1.3.1 Graph View

All the earlier figures depicting the network were created manually to meet cosmetic expectations. This approach is not viable for actual solutions because it is too tedious and error prone. Instead, the system can automatically generate an image of the network, as in Figure 26.

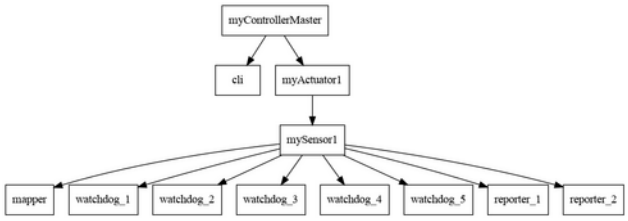

Figure 26: Network Graph View

This visualization is the first step in verifying that the model is structurally correct before simulating it. It helps identify obvious missing components or misconnections.

3.1.3.2 XML View

The graphical representation is helpful for visual inspection, but it omits the configuration details. The system can also generate a corresponding XML (Extensible Markup Language) representation. XML is a standard format that any web browser can import and then automatically format in a meaningful way with indentation and color, as in Figure 27. This feature gives students exposure to using external tools for analysis.

```
'<network>
 ▼<controller id="myControllerMaster" type="MyControllerMaster">
  ▼<components>
   ▼<component id="cli" controllerID="myControllerMaster">
    ▼<groups>
       <group id="interface"/>
       <group id="all"/>
     </groups>
    </component>
   ▼<actuator id="myActuator1" controllerID="myControllerMaster">
    ▼<groups>
       <group id="actuator"/>
       <group id="all"/>
     </groups>
    ▼<sensor id="mySensor1" controllerID="myControllerMaster">
     ▼<groups>
        <group id="sensor"/>
        <group id="all"/>
      </groups>
     ▼<mapper type="interpolator">
      ▼<interpolator type="spline">
       ▼<map>
          <entry valueIndependent="0.0" valueDependent="5.0"/>
          <entry valueIndependent="10.0" valueDependent="100.0"/>
        </map>
       </interpolator>
      </mapper>
     ▼<watchdogs>
        <watchdog type="acceleration" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
        <watchdog type="acceleration" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
        <watchdog type="band" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
        <watchdog type="notch" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
        <watchdog type="notch" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
        <watchdog type="notch" thresholdLow="0.0" thresholdHigh="5.0" grace="3"/>
        <watchdog type="low" threshold="0.0" grace="3"/>
        <watchdog type="high" threshold="0.0" grace="3"/>
        <watchdog type="high" threshold="0.0" grace="3"/>
        <watchdog type="high" threshold="0.0" grace="3"/>
        <watchdog type="high" threshold="0.0" grace="3"/>
      </watchdogs>
     ▼<reporters>
        <reporter type="change" deltaThreshold="2"/>
        <reporter type="frequency" reportingFrequency="3"/>
      </reporters>
    </sensor>
   </actuator>
  </components>
 </controller>
</network>
```

Figure 27: Network XML View

3.1.3.3 JSON View

Similarly, many tools (including browsers) work with another industry-standard format, JSON (JavaScript Object Notation). The system can export the same information in this representation, as in Figure 28.

```
{
  "network": {
    "controller": {
      "components": {
        "component": {
          "groups": {
            "group": [
              {
                "_id": "interface"
              },
              {
                "_id": "all"
              }
            ]
          },
          "_id": "cli",
          "_controllerID": "myControllerMaster"
        },
        "actuator": {
          "groups": {
            "group": [
              {
                "_id": "actuator"
              },
              {
                "_id": "all"
              }
            ]
          },
          "sensor": {
            "groups": {
              "group": [
                {
                  "_id": "sensor"
                },
                {
                  "_id": "all"
                }
              ]
            },
```


Figure 28: Network JSON View

3.1.3.4 State View

A message goes through a life cycle from creation to processing to destruction. It can be important for analysis (especially debugging) to know which state a message is in at any point. The system can automatically generate state diagrams, as in Figure 29.

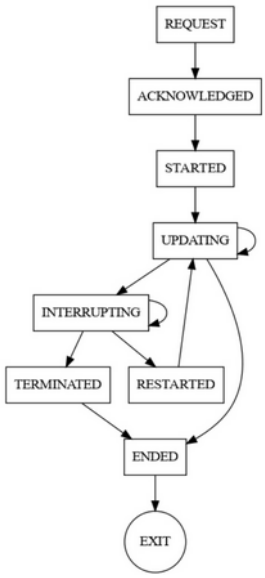

Figure 29: Actuator State Diagram

3.1.3.5 Network View

Similar to the state view, the route a message takes across the network can be important. The system can automatically generate swim-lane diagrams to depict message traffic, as in Figure 30. In this example, each of five components occupies a column. Message traffic is depicted as an unbroken sequence of arrows from creation to destruction.

Figure 30: Network Traffic View

3.1.3.6 Log View

The system runs a continuous time-stepped simulation. This execution generates a text-based log file with the state data from the simulation and from every reporter at each time step. The CSV (comma-separated values) format can be imported directly into Excel or any similar tool for viewing and manipulation after the simulation completes. Figure 31 provides a more detailed view of Figure 30.

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		

Figure 31: Log Network View

Figure 32 provides a view of an actuator in motion. It reports the position and speed at each time step, as well as a more detailed view of Figure 29.

time	position	velocity	comment
0.33	2	0	StateAscendingLeadin
0.34	2.1	0.1	StateAscendingLeadin
0.35	2.3	0.2	StateAscendingLeadin
0.36	2.6	0.3	StateAscendingLeadin
0.37	3	0.4	StateAscendingLeadin
0.38	3.5	0.5	StateAscendingLeadin
0.39	4.1	0.6	StateAscendingLeadin
0.40	4.8	0.7	StateAscendingLeadin
0.41	5.6	0.8	StateAscendingLeadin
0.42	6.5	0.9	StateAscendingLeadin
0.43	7.5	1	StateAscendingLeadin
0.44	8.6	1.1	StateAscendingLeadin
0.45	9.8	1.2	StateAscendingLeadin
0.46	11.1	1.3	StateAscendingLeadin
0.47	12.5	1.4	StateAscendingLeadin
0.48	14.4	-1.4	StateAscendingLeadin
0.49	15.8	-1.2	StateAscendingLeadout

Figure 32: Log Actuator View

3.1.3.7 Graph View

Once the actuator state data is in Excel, all of its graphing capabilities are available to visualize what is happening. Figure 33, for example, shows eight[3] flight control surfaces starting from a neutral position (0 degrees on the *y*-axis) at callout (1), moving at different rates to +90 degrees over time on the *x*-axis until meeting up at (2), and then each half splitting to +20 and −20 degrees, respectively. It is essential to be able to plot simultaneous actions in a complex system to verify that they occur correctly and to report these results.

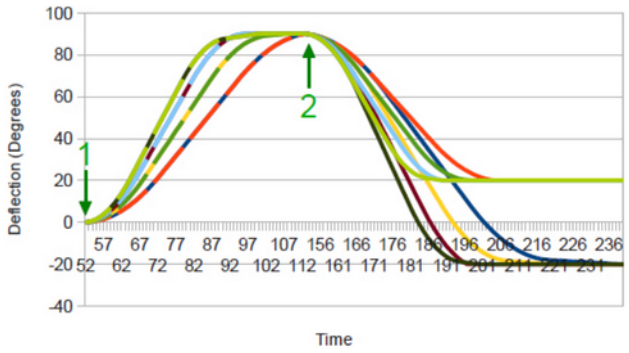

Figure 33: Graph View 1

Figure 34 shows the same components performing a similar operation with more diversity in the states. Visual analysis is intuitive because the eye is often easily drawn to discrepancies, such as lack of symmetry, discontinuity, or misaligned convergence points.

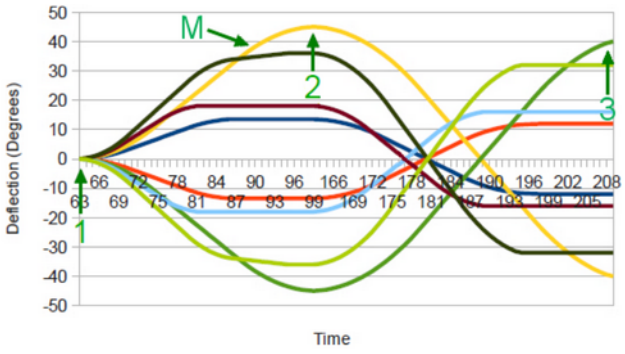

Figure 34: Graph View 2

3.1.3.8 Test and Evaluation

The final process of test and evaluation allows students to play the role of an end user. The instructor provides a set of test requirements, such as:

> *Create a linear actuator and manipulate it as follows: Starting at 0 degrees, command it to −45 degrees; upon arrival, command it to +45 degrees; upon arrival, command it to 0 degrees; upon arrival, command it to −30 degrees, but at −15 degrees interrupt it with a command to +45 degrees, and allow it to arrive.*

For each test, they must address the seven elements stated verbatim in the assignment:

1. The rationale behind the test; i.e., what is it testing and why we care.
2. A general English description of the initial conditions of the test.
3. The commands for (2), which must appear in a standalone form that could be directly copied into a text file to reproduce the test without manual intervention. Do not cross-reference other tests.
4. A brief English narrative of the expected results of executing the test. Proper testing discipline expects that you establish this *before* running the test.
5. At least one representation of the actual results. The form is your choice.
6. A brief discussion on how the actual results differ from the expected results, if at all.
7. A suggestion for how to extend this test to cover related aspects not required here.

Figure 35 shows an appropriate visualization for (5) in this example.

[3] Multicolored lines are exactly on top of each other.

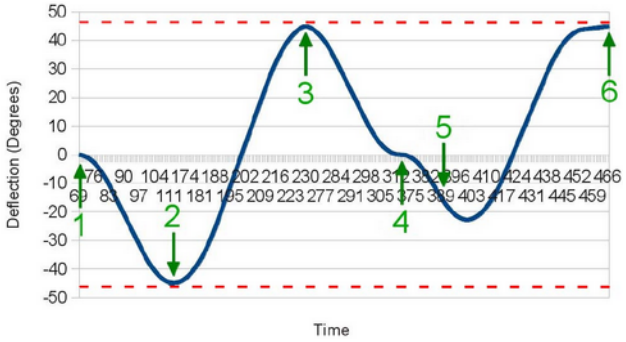

Figure 35: Test and Evaluation Result

The callouts refer to the following key states:

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

The callouts and artwork in the graph are manual processes. Students need to learn how to present complex data in an appropriate form, which is expected in industry.

The `@RUN` command is especially convenient here to keep everything organized. It allows students to put each test definition in a separate script file and to indicate where to output the results.

4 Examples

The following two examples provide an overview of actual projects that students worked on for an entire 10-week quarter in a junior-level introductory course on software engineering.

4.1 Fly-by-Wire Aircraft Control System

The airplane in Figure 11 depicts the actuator-driven components that would allow it to fly. Nevertheless, this project is not a flight simulator, so the aircraft remains stationary on the ground. However, it executes the same general actions that it would in flight. Its control system in Figure 12 is relatively simple in that it just moves the components appropriately to demonstrate aspects of interest. The students used the text-based inputs from Section 3.1.2.1.3 to control this behavior. Control is not directly tied to a realistic user interface like a joystick, but this capability will be provided eventually.

Figure 36 shows a three-dimensional view of the flight control surfaces in red. This part is currently provided by an external visualization tool, so it is not automatically available to the students. The instructor must build it manually and feed it the appropriate data after the simulation completes. This visualization will eventually be part of the built-in views in Section 3.1.3.

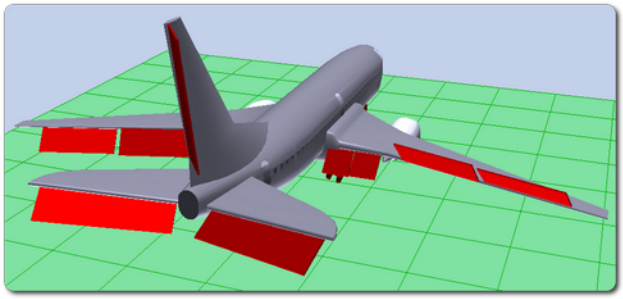

Figure 36: Airplane 3D View

The control surfaces are kept simple because the course is not about aerospace engineering. However, the capabilities of the system allow for much more complexity, such as the full complement for a real-world airliner in Figure 37.

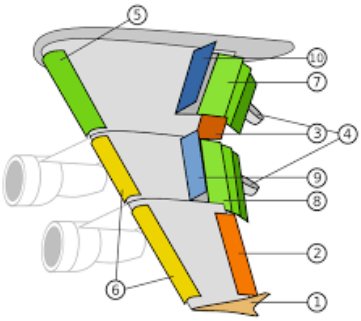

Figure 37: Flight Control Surfaces [3]

Not only does this model have far more components, they also behave in far more complex ways. Figure 36 depicts simple “barn door” surfaces with no mechanical interdependencies. Figure 38, on the other hand, represents the multistage actions necessary to deploy a sequence of components appropriately. This model would be overkill for a software engineering course, but it is appropriate for one on modeling and simulation and aerospace engineering, for example.

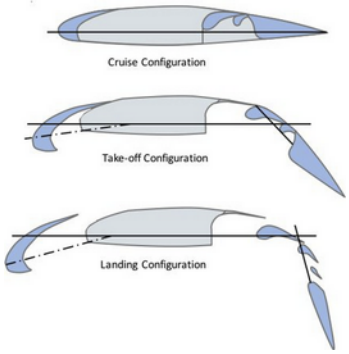

Figure 38: Slat and Flap Deployment [11]

4.2 Heavy Construction Equipment

Figure 39 depicts a similar project with different mechanisms. Here the components are articulated members of the excavator, such as the bucket and multilinkage boom. The model represents hydraulic, pneumatic, and electrical systems. As with the airplane, the goal is not to perform construction activities, but rather to demonstrate the underlying concepts.

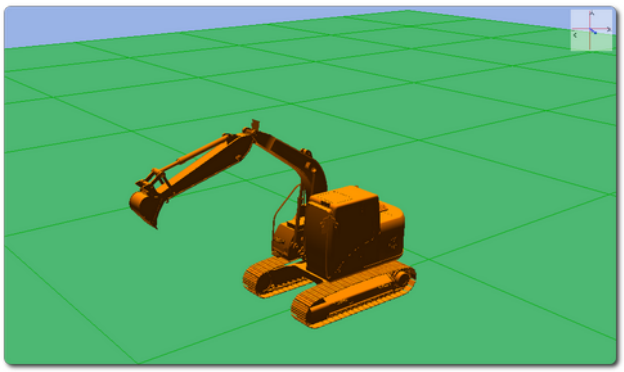

Figure 39: Excavator

5 Future Work

From the technical side, a better three-dimensional viewer is desirable. Examples currently require extensive graphics programming in an external visualization tool. Simplifying this process and integrating it into this system would accommodate a richer breadth and depth of examples with less effort.

From the pedagogical side, conveniently packaged resources need to be made available. There is currently no way for anyone outside the author's classroom environment to use this system. Having it available for download, along with examples and tutorials, would make it useful to others.

6 Conclusion

This system is designed around the needs of students. It provides them with realistic but manageable theoretical and practical experiences in using modeling and simulation as part of software development. It also connects many of the pieces of their studies into a more coherent form for better understanding and application. The system is also convenient for the instructor to be able to walk students through various stages of development, set up experiments, and evaluate performance.

References

[1] Brown, H. *507 Mechanical Movements*. Martino Fine Books, 2013.

[2] Charette, R. "How Software is Eating the Car." IEEE Spectrum, 7 June 2021.

[3] Creative Commons noncommercial license.

[4] Cunningham, P., Matusovich, H., Hunter, D., McCord, R. "Teaching metacognition: Helping engineering students take ownership of their own learning." 2015 IEEE Frontiers in Education Conference, El Paso, TX, USA, 2015, pp. 1-5.

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

[6] Hillhouse, G. *Engineering in Plain Sight*. No Starch Press, 2022.

[7] Lauesen, S. "IT Project Failures, Causes and Cures." IEEE Access, vol. 8, pp. 72059-72067, 2020.

[8] Ossamah, A., Meshari, A., Yazed, A., and Norah, A. "Cloud Based Cyber Physical System for Factory Automation." 2020 IEEE 6th World Forum on Internet of Things, New Orleans, LA, USA, 2020, pp. 1-7.

[9] Soliman, F. *Mechatronics: Multidisciplinary Engineering*. Lambert Academic Publishing, 2017.

[10] Usually attributed to Edsger Dijkstra as a direct quote but actually a paraphrase.

[11] Variations on this figure have been floating around the internet for years. There is no clear attribution anymore.

Author Biography

DAN TAPPAN is a professor and director of computer science and electrical engineering at Eastern Washington University. He has been a professor for over 18 years and before that 10 years as a DoD civilian at White Sands Missile Range and Aberdeen Proving Ground mostly doing software engineering and modeling and simulation of flight and weapons systems. His main research areas are software and hardware systems engineering, especially for aviation and military applications with embedded systems and mechatronics; modeling, simulation, visualization, and analysis; intelligent systems/artificial intelligence; and computer science and engineering education.