Regular Paper

Comparison of Dynamic System Modeling Methods

A. Terry Bahill* and Ferenc Szidarovszky

Systems and Industrial Engineering, University of Arizona, Tucson, AZ 85721-0020

Received 24 January 2008; Revised 26 March 2008; Accepted 26 June 2008, after one or more revisions
Published online 3 October 2008 in Wiley InterScience (www.interscience.wiley.com)
DOI 10.1002/sys.20118

ABSTRACT

This paper compares state-equation models to state-machine models. It compares continuous system models to discrete system models. The examples were designed to be at the same level of abstraction. This paper models these systems with the following methods: the state-space approach of Linear Systems Theory, set-theoretic notation, block diagrams, use cases, UML diagrams and SysML diagrams. This is the first paper to use all of these modeling methods on the same examples. © 2008 Wiley Periodicals Inc. Syst Eng 12: 183–200, 2009

Key words: model-based systems engineering; UML; SysML; linear systems theory

1. INTRODUCTION

System design can be requirements based, function based, or model based. Model-based system engineering and design has an advantage of executable models that improve efficiency and rigor. One of the earliest developments of this technique was Wymore's [1993] book entitled *Model-Based System Engineering*, although the phrase *Model-Based System Design* was in the title and topics of Rosenblit's [1985] Ph.D. dissertation. Model-based systems engineering depends on having and using well-structured models that are appropriate for the given problem domain.

There are two types of systems: static and dynamic. In a *static system*, the outputs depend only on the present values of the inputs, whereas in a *dynamic system* the outputs depend on the present and past values of the inputs [Botta, Bahill, and Bahill, 2006]. In computer design, these two basic types of systems are called combinational and sequential. Combinational systems do not require memory devices; hence they are called memoryless. Sequential systems require memory to capture the state behavior. In combinatorial systems, the output depends only on the present inputs, whereas in sequential systems the output depends on the sequence of previous inputs. In mechanical engineering, these two types of systems are called static and dynamic. Static systems are described with algebraic equations and the outputs depend only on the present values of the inputs, whereas dynamic systems are described with differential or difference equations and the system behavior depends on the history of the inputs. This paper only considers dynamic systems.

*Author to whom all correspondence should be addressed (e-mail: terry@sie.arizona.edu; szidar@sie.arizona.edu).

The purpose of this paper is to compare and contrast several systems engineering methods for modeling two different types of dynamic systems and eventually to explain systems composed of both types of systems.

The motivation for this paper was to show how the SysML diagrams fit in with traditional modeling methods. The paper is intended to assist a modeler in understanding the pros and cons of different modeling methods as they apply to different classes of problems.

Some dynamic systems are modeled best with *state equations* while others are modeled best with *state machines*. State-equation systems are modeled with differential or difference equations. For example, a baseball's movement can be modeled with state equations for position, linear velocity, and angular velocity all as functions of time. In contrast, state-machine systems focus less on physical variables and more on logical attributes. Therefore, these systems have memory and are modeled with finite state machines. Most digital computer systems are modeled best with state machines.

Newton [1687] was the first to apply equations to model and understand state-equation systems. Much later, during World War II, the use of feedback produced a dramatic improvement in system design. In this period, because the primary purpose of dynamic system design was to stabilize the firing of guns on naval ships, the field was called fire control systems. After the war the frequency response techniques for solving differential equations were developed. Later, in the 1960s, the state-space techniques bloomed and they are still the dominant method [Szidarovszky and Bahill, 1998]. Typical state-equation systems include analog computers as well as electrical, mechanical, hydraulic, pneumatic, thermal, economic, chemical and biological systems.

Turing [1936] was the first to model and explain state-machine systems. His Turing Machine (a state machine with a memory) could solve all solvable sequential logic problems. The field came to be known as *finite state machines*. The digital computer is the preeminent example of state-machine systems.

The methods that will be used in this paper to model these systems include the state-space approach of Linear Systems Theory (for both continuous and discrete systems) [Szidarovszky and Bahill, 1998], Wymorian set-theoretic notation [Wymore, 1993], block diagrams, use cases, UML diagrams [OMG UML, 2007; Fowler, 2004] and SysML diagrams [OMG SysML™, 2007 SysML, 2007].

In designing complex systems, such as an airplane or an automobile, it might be necessary to use all of these methods together. Some parts might be continuous while others are discrete. Some might be modeled best with UML diagrams and others with block diagrams, depending on the problem and the backgrounds of the engineers involved. So, although one system could use all of these methods, in this paper we will discuss each method separately.

Section 2 of this paper examines state-equation systems and uses as an example a simple spring, mass, dashpot system. Section 3 examines state-machine systems and uses as an example a simple spelling checker. Most real-world systems are modeled as either state-equation systems or state-machine systems. However, there are physical systems where both types of models are useful together: for example, systems using water, because the heat flow equations are different depending on whether H_2O is in its liquid, solid or gas state. Section 4 presents a problem where the selection of the preferred modeling approach is not clear.

2. STATE-EQUATION SYSTEMS

First, we will look at an example from the field of Linear Systems Theory. In particular, we will use a method called the state-space approach.

2.1. Continuous Systems

This section is based on Szidarovszky and Bahill [1998] and Ogata [2004]. Consider the mechanical system illustrated in Figure 1. Assume an ideal spring, mass and dashpot, and frictionless wheels. Assume initial conditions are zero and do a small signal analysis. Create a state-space model for this system.

In general, the systems engineer gets models from the domain experts. For this system, assume we are given these models: $f_k = kx$, $f_b = b\,dx/dt = b\dot{x}$, and $f_m = ma = m\ddot{x}$. We will use SI units: the spring constant (k) has units of N/m, the damping coefficient (b) has units of N·s/m, and the mass (m) has units of kg.

Now, let us define the input, output and states. The force $f(t)$ is applied to the system and it produces a displacement $x(t)$. Define $f(t)$ as the input u and the displacement x as the output. From the diagram $f(t) = m\ddot{x} + b\dot{x} + kx$, which can be rewritten as $\ddot{x} = (1/m)(-kx - b\dot{x} + f(t))$. The system is of second order, so we need to select two state variables. Let us choose the position x and the velocity $\dot{x}$. So that we have

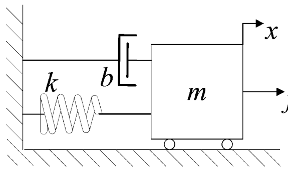


Figure 1. SpringMassDashpot system.

$u = f(t) =$ input,
$x_1 = x =$ position,
$x_2 = \dot{x} =$ velocity.

The state equations then become

$$\dot{x}_1 = x_2,$$

$$\dot{x}_2 = -\frac{k}{m}x_1 - \frac{b}{m}x_2 + \frac{1}{m}u.$$

For the output variable we choose

$$y = x_1 = \text{output}$$

Rewriting the state equations and the output equation in matrix form, we obtain

$$\begin{pmatrix}\dot{x}_1\\ \dot{x}_2\end{pmatrix} = \begin{pmatrix}0 & 1\\ -\frac{k}{m} & -\frac{b}{m}\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} + \begin{pmatrix}0\\ \frac{1}{m}\end{pmatrix}u,$$

$$y = (1 \quad 0)\begin{pmatrix}x_1\\ x_2\end{pmatrix}.$$

These equations are in the standard form of

$$\dot{\mathbf{x}} = \mathbf{Ax} + \mathbf{Bu},$$

$$\mathbf{y} = \mathbf{Cx} + \mathbf{Du}$$

with

$$\mathbf{A} = \begin{pmatrix}0 & 1\\ -\frac{k}{m} & -\frac{b}{m}\end{pmatrix},\quad \mathbf{B} = \begin{pmatrix}0\\ \frac{1}{m}\end{pmatrix},\quad \mathbf{C} = (1 \ \ 0),\quad \mathbf{D} = 0.$$

2.1.1. Block Diagrams

Block diagrams, as shown in Figure 2, are used to illustrate state-equation systems [Buede, 2000].

2.2. Discrete-Time Systems

Unlike the previous continuous-time system, for physical systems containing computers we are only interested in the system's behavior at discrete time intervals. Many other physical systems can also be modeled as discrete systems [Wymore, 1993], if we assume a fast enough sampling rate. For discrete systems, we have as a good approximation

$$\frac{\mathbf{x}(t+h) - \mathbf{x}(t)}{h} = \mathbf{Ax}(t) + \mathbf{B}u(t),$$

implying that

$$\mathbf{x}(t+h) = \mathbf{x}(t) + h\,(\mathbf{Ax}(t) + \mathbf{B}u(t)).$$

That is,

$$\mathbf{x}(t+h) = (\mathbf{I} + h\mathbf{A})\mathbf{x}(t) + h\mathbf{B}u(t),$$

in which case we have

$$\mathbf{x}(t+h) = \begin{pmatrix}1 & h\\ -\frac{hk}{m} & 1-\frac{hb}{m}\end{pmatrix}\mathbf{x}(t) + \begin{pmatrix}0\\ \frac{h}{m}\end{pmatrix}u(t),\quad \mathbf{x}(0) = \begin{pmatrix}0\\ 0\end{pmatrix}$$

$$t \geq 0.$$

$$y(t) = (1 \quad 0)\mathbf{x}(t),$$

If we write this as

$$\mathbf{x}(t+h) = \mathbf{Px} + \mathbf{Q}u$$

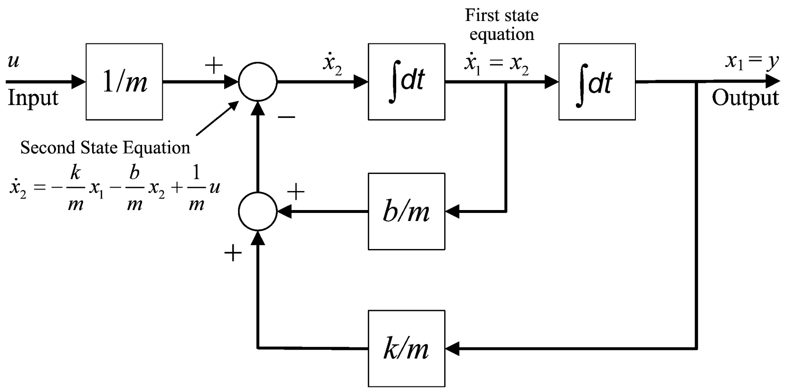


Figure 2. Block diagram of the SpringMassDashpot system.

and if

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix},$$

then in general

$$\mathbf{P} = \begin{pmatrix} 1 + ha_{11} & ha_{12} \\ ha_{21} & 1 + ha_{22} \end{pmatrix}, \quad \mathbf{Q} = \begin{pmatrix} hb_1 \\ hb_2 \end{pmatrix}.$$

There are also systems that are inherently discrete without computer control. For instance, if you borrowed B_0 dollars from a bank and agreed to pay interest of r percent per month and make n monthly payments of amount P, then the balance B that you will owe can be written as $B(t + h) = (1 + r)B(t) - P(t)$, where h is 1 month.

2.2.1. Stability of Continuous and Discrete Systems

It is well known that if you start with a continuous system and add sensors, a digital processor, and actuators to make it discrete, then the discrete system will be closer to instability than the continuous system.

Consider a continuous system

$$\dot{\mathbf{x}}(t) = \mathbf{Ax}(t) + \mathbf{Bu}(t)$$

and its discrete counterpart

$$\mathbf{x}(t + h) = \mathbf{x}(t) + h[\mathbf{Ax}(t) + \mathbf{Bu}(t)].$$

For the continuous system the coefficient matrix is $\mathbf{A}$, while for the discrete system it is $\mathbf{I} + h\mathbf{A}$, where h is the step size in the time scale. Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the eigenvalues of $\mathbf{A}$, then the eigenvalues of the corresponding discrete system are $1 + h\lambda_1, 1 + h\lambda_2, \ldots, 1 + h\lambda_n$. Assume now that $\lambda = \alpha + j\beta$ is an eigenvalue of $\mathbf{A}$. The condition for asymptotic stability is that α be negative. The corresponding condition for the discrete system is that $1 + h(\alpha + j\beta)$ be inside the unit circle, which happens if and only if $(1 + h\alpha)^2 + (h\beta)^2 < 1$.

In summary, to be stable

(i) for a continuous system $\alpha < 0$,
(ii) for a discrete system $(1 + h\alpha)^2 + (h\beta)^2 < 1$.

Notice that (ii) implies (i). If (ii) is true, then $|1 + h\alpha| < 1$, that is, $-1 < 1 + h\alpha < 1$, which can be written as $-2/h < \alpha < 0$. However, notice that (i) does not imply (ii). So the asymptotic stability of the discrete system implies that the corresponding continuous system is also asymptotically stable. However, the asymptotic stability of the continuous system does not imply asymptotic stability of the corresponding discrete system for all h.

We usually choose h to be small. But how small should it be? Assume that $\alpha < 0$. The discrete system will be asymptotically stable if h is sufficiently small. Condition (ii) can be rewritten as $1 + 2h\alpha + h^2(\alpha^2 + \beta^2) < 1$, that is, $h(\alpha^2 + \beta^2) < -2\alpha$ or $h < -2\alpha/(\alpha^2 + \beta^2)$, where the right-hand side is positive.

In the case of nonlinear systems we have the same conclusions, except instead of the matrix $\mathbf{A}$, we use the Jacobian of the continuous system.

2.3. Wymorian Notation

Next, we can put this SpringMassDashpot system into Wymorian set-theoretic notation [Wymore, 1993]. Wymore uses, for example, the symbol p2 for a particular value of the second input, x2 for a particular value of the second state variable, and y2 for a particular value of the next state of the second state variable, etc. Thus, Wymore's y is not the same as the output y in the above equations. Comments are enclosed in /* markers */. Let us name the system Zsmd. Then

```
Zsmd = {SZsmd, IZsmd, OZsmd, NZsmd, RZsmd}
  where
  SZsmd = RLS ^ 2,   /*There are 2 state variables
     and they take values from the set of real num-
     bers, ℝ .*/
  IZsmd = RLS,       /*The input values will be real
     numbers.*/
  OZsmd = RLS,    /*The output values will be real
     numbers.*/
  NZsmd = {((x, p), y): x ∈ SZsmd; p ∈ IZsmd; y
     ∈ SZsmd;
       if x = (x1, x2) then
       y = [y1, y2] = [(x1 + hx2), (- hkx1/m +
          (1 – hb/m)x2 + hp/m)]}, /*The state
          equation. Remember p is a value of
          the input.*/
  RZsmd = S1Zsmd.   /*The output is the first state
     variable.*/
```

Please note that with Wymorian notation a system can be described with a dumb typewriter: It does not require a computer, a word processor, a drawing program, an equation editor, subscripts, superscripts, or even italic and boldface fonts.

2.4. UML Representation

There are no appropriate UML diagrams for state-equation systems.

2.5. OMG SysML™ Model

The OMG Systems Modeling Language (OMG SysML™) was created as systems engineering extensions to the Unified Modeling Language (UML). SysML reduces UML's software-centric restrictions and adds two new types of diagrams: Requirement diagrams to help manage requirements, and Parametric diagrams to help with performance and quantitative analysis. SysML was designed to model large complex systems such as an automobile. Requirement diagrams are used to efficiently capture functional, performance and interface requirements, and Parametric diagrams are used to precisely define performance and mechanical constraints such as maximum acceleration, curb weight, air conditioning capacity, and interior cabin noise. SysML also enhanced the semantics and greatly increased the usage of UML's Activity diagram, Block Definition diagram, and Internal Block diagram.

This section is written at the nuts and bolts level. It shows complete, detailed use of SysML constructs. However, it does not present top-down descriptions of large complex systems. It does not address the massive concurrency and interfaces of large complex systems. Most examples of large complex systems do not give a complete model at the nuts and bolts level. They usually discuss the top-level and then drive down to the bottom on only a few traces: They only show slivers of the whole system. A nice example that spans the whole hierarchy is the Hybrid SUV example [OMG SysML, 2007; Friedenthal, Moore, and Steiner, 2007]. It is impossible to show a complete example of a highly complex highly parallel system. Therefore, this paper does not discuss the hierarchy of systems: This paper models only simple systems, but its coverage is complete. All of the examples are deliberately at the same level of abstraction [Mellor et al., 2004; Bahill et al., 2008].

SysML is supported by the OMG XMI 2.1 model interchange standard for UML 2 modeling tools. It has also been conceptually aligned with the ISO 10303 STEP AP233 (http://www.ap233.org/) data interchange standard for systems engineering tools. SysML supports model management concepts that include views and viewpoints, which extend UML's capabilities and are architecturally aligned with IEEE-Std-1471-2000.

The SysML can use state equations to specify constraints on properties of the system and its environment using the block definition diagram (**bdd**) and the parametric diagram (**par**) [OMG SysML, 2007].

First, we define the input and output of our SpringMassDashpot system. The input is a flow port: energy flows through this port. It is the force applied to the mass. The output is the position of the mass: It is a standard port. Inputs and outputs are often shown on an internal block diagram (**ibd**).

Next, we want to show the components of our SpringMassDashpot system. We do this in a block definition diagram, as shown in Figure 3. The block for each component can be further elaborated to shows its functions, interfaces and properties. In Figure 3, we only see examples of some of the properties of the blocks, such as the position of the Mass. The applied force is labeled Fixed Reference.

In SysML diagrams the header has four fields: the type of diagram (in this case, **bdd** standing for block definition diagram), the type of model element that the diagram represents (package), the name of the model element described in the diagram (Mechanical System), and the fourth field is user defined and can, for example, state the purpose of the diagram (Components of SpringMassDashpot system). The arrow with a black diamond head indicates a composition relationship.

Table I explains the roles and abbreviations shown in Figures 3–7.

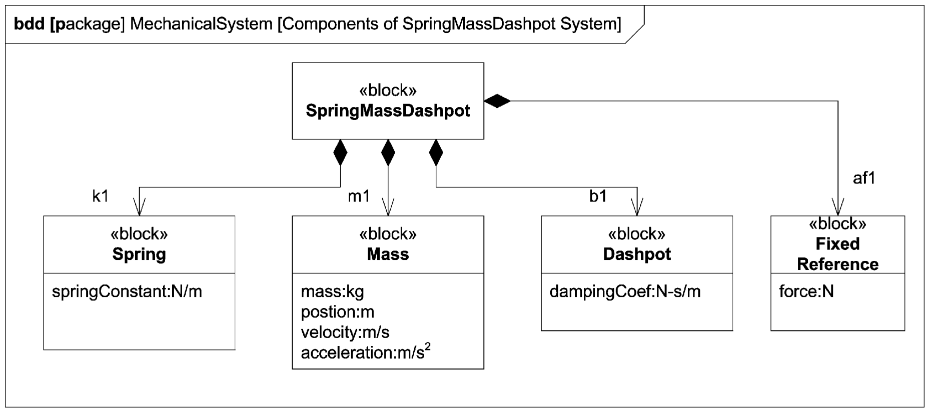


Figure 3. Block definition diagram (**bdd)** showing the components of the Mechanical System.

Table I. Explanation of Role Names

Usage name, or part name, or Role	Explanation
k1	Role name of one particular spring
m1	Role name of one particular mass
b1	Role name of one particular dashpot
af1	Role name of one particular applied force
firstEq	First state equation
secondEq	Second state equation
outputEq	Output equation
smd	Role name for the SpringMassDashpot block

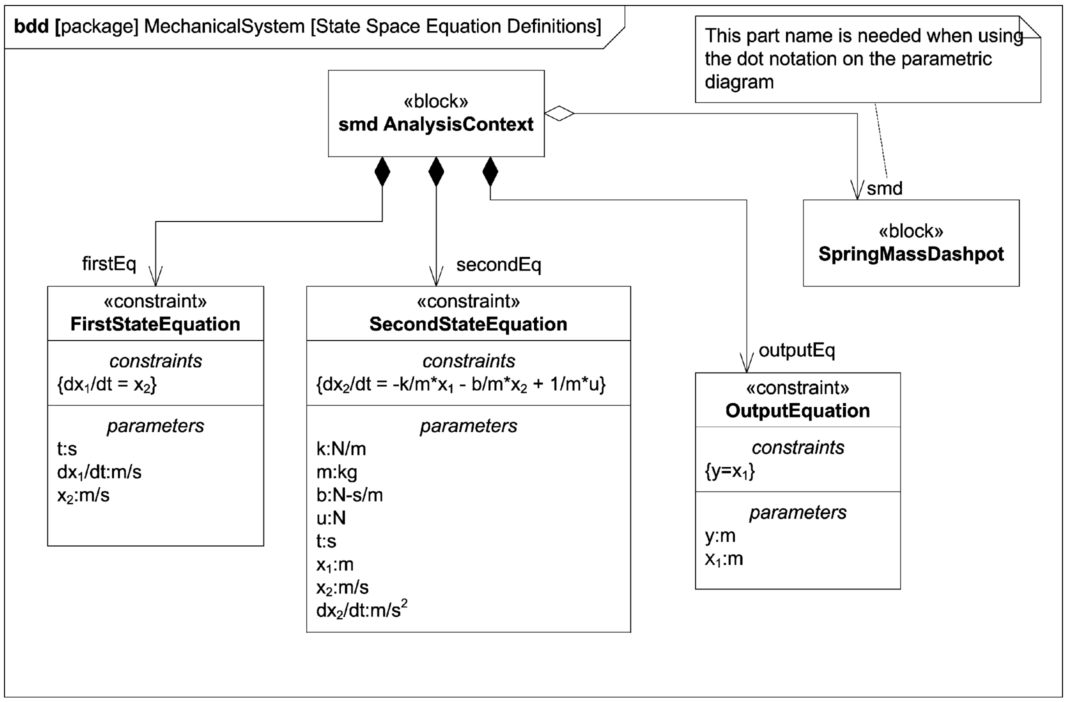
Next, we want to present the constraints or equations that describe our mechanical system.

The **bdd** of Figure 4 defines the state equations in the <<constraint>> blocks. These state equations for this smd Analysis Context may have originally been specified in an analysis library. This enables the same equations to be reused for many different contexts. The arrows with solid (black) diamond endings are composition relationships, which means that the smd Analysis Context comprises the FirstStateEquation, the SecondStateEquation, and the OutputEquation. The arrow with a white (open) diamond head indicates an aggregation relationship between the smd AnalysisContext and the SpringMassDashpot block from Figure 3. In the parameter lists, on the left of the colon are the parameters, or names as they appear in the equation, and on the right of the colon are their units.

Now we want to show how these general equations can be instantiated to a particular SpringMassDashpot system. The top part of Figure 5 is based on the blocks of Figure 3, and the bottom part is based on the constraints of Figure 4. The interconnections show the bindings of the parameters.

The parametric diagram of Figure 5 shows how the parameters from each of the equations can be bound together to create a complex network of equations. The constraint blocks that were specified in the **bdd** of Figure 4 are bound to one another to create a network of the state equations that bind the parameters together. Although not explicitly shown here, the parametric diagram enables the properties of the SysML design models, such as the spring constant of the spring, to be unambiguously associated with the parameters of the equations. In this way, the "system design model" and the "system analysis model" can be fully integrated. The full power of this approach becomes apparent as many different analysis models related to performance, reliability, mass properties, etc. are integrated with the system design model that may be defined via activity diagrams, internal block diagrams, etc.

Each property that is constrained by the parametric equations may include values, including probability distributions, with units and dimensions. The dimension represents the fundamental quantity type. For example, the width of an object may have a dimension of



bdd [package] MechanicalSystem [State Space Equation Definitions]
This part name is needed when using the dot notation on the parametric diagram
«block» smd AnalysisContext
smd
«block» SpringMassDashpot
firstEq
secondEq
outputEq
«constraint» FirstStateEquation
constraints
{dx1/dt = x2}
parameters
t:s
dx1/dt:m/s
x2:m/s
«constraint» SecondStateEquation
constraints
{dx2/dt = -k/m*x1 - b/m*x2 + 1/m*u}
parameters
k:N/m
m:kg
b:N-s/m
u:N
t:s
x1:m
x2:m/s
dx2/dt:m/s2
«constraint» OutputEquation
constraints
{y=x1}
parameters
y:m
x1:m

Figure 4. Block definition diagram (**bdd**) defining constraints for the mechanical system.

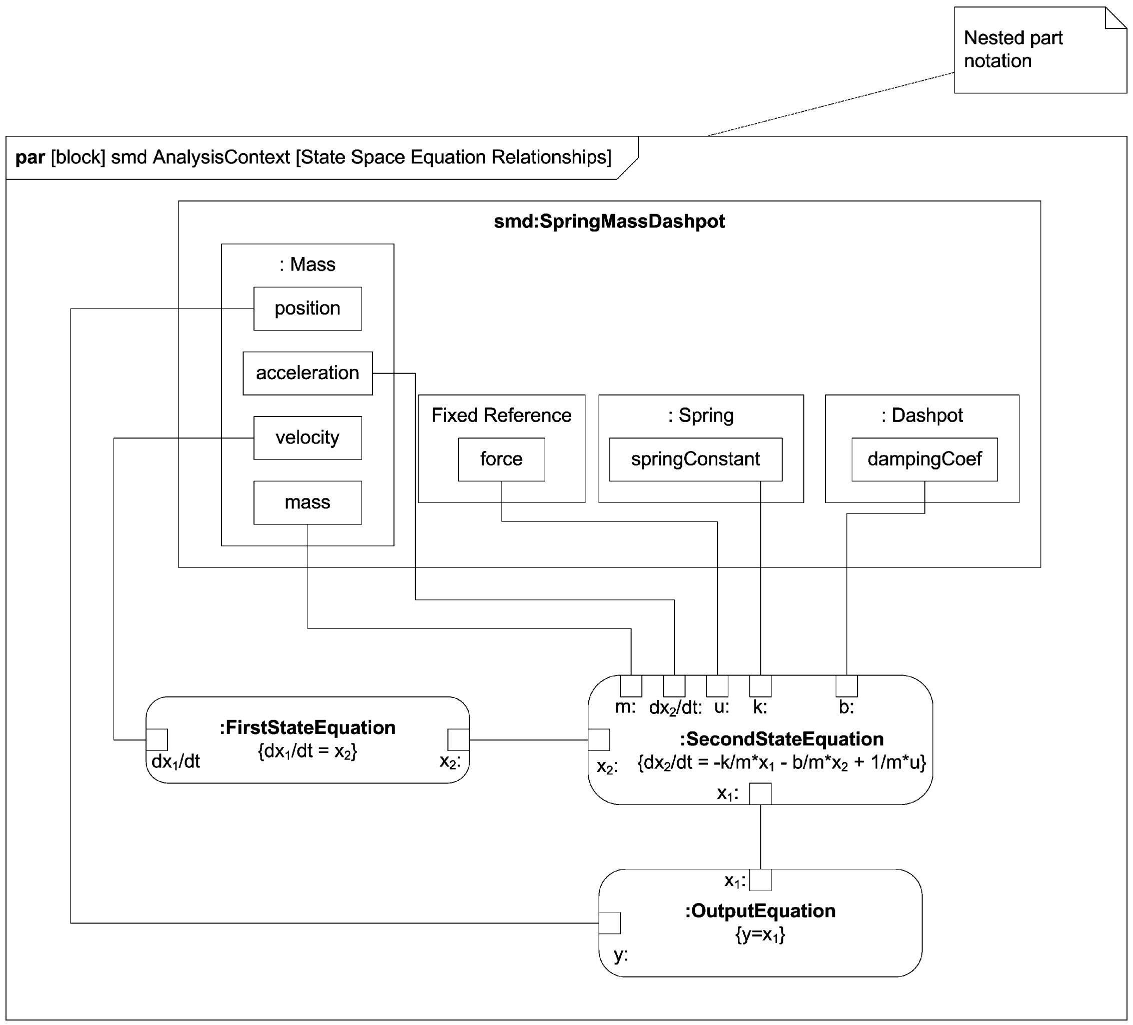


Figure 5. Parametric diagram (**par**) using nested part notation showing the state-space equations for a particular SpringMass-Dashpot system.

length and units of feet or meters. The assignment of units and dimensions is accomplished by typing the property with a value type that includes the applicable dimensions and units. A fully compliant SysML tool can check to confirm the compatibility of the units and dimensions of its properties, and avoid significant issues that have occurred when the units have been improperly matched.

A parametric diagram is not an equation solver like MATLAB or Mathematica. It does not define independent and dependent variables. It merely shows bindings, or equality. For example, the parametric diagram of Figure 5 shows that dx_1/dt is equal to the velocity of the mass. It does not tell you how to solve for it. In contrast, an equation solver might state something like, to update x_1 start with the initial position and integrate x_2 using Adams-Moulton or Runge-Kutta integration routines.

Figure 6 presents an alternative parametric diagram with the same content but a different notation. Figure 5 uses nested part notation. For example, the parameter b in the second state equation is bound to the dampingCoef, which is drawn as a box inside of the Dashpot, which is in turn drawn as a box inside of the SpringMassDashpot, whereas Figure 6 uses dot notation, which specifies the path down through the nested part hierarchy to the property. For example, "smd.b1.dampingCoef" specifies that the dampingCoef is a property of the Dashpot "b1", which is part of the SpringMassDashpot "smd". The abbreviations smd and b1 were introduced in Figures 3 and 4. These part names are separated with periods or "dots."

State equations are always tightly coupled: They are usually thought of as a matrix. In SysML, constraints can be composed of more than one equation. Therefore, our **bdd** could be composed as shown in Figure 7.

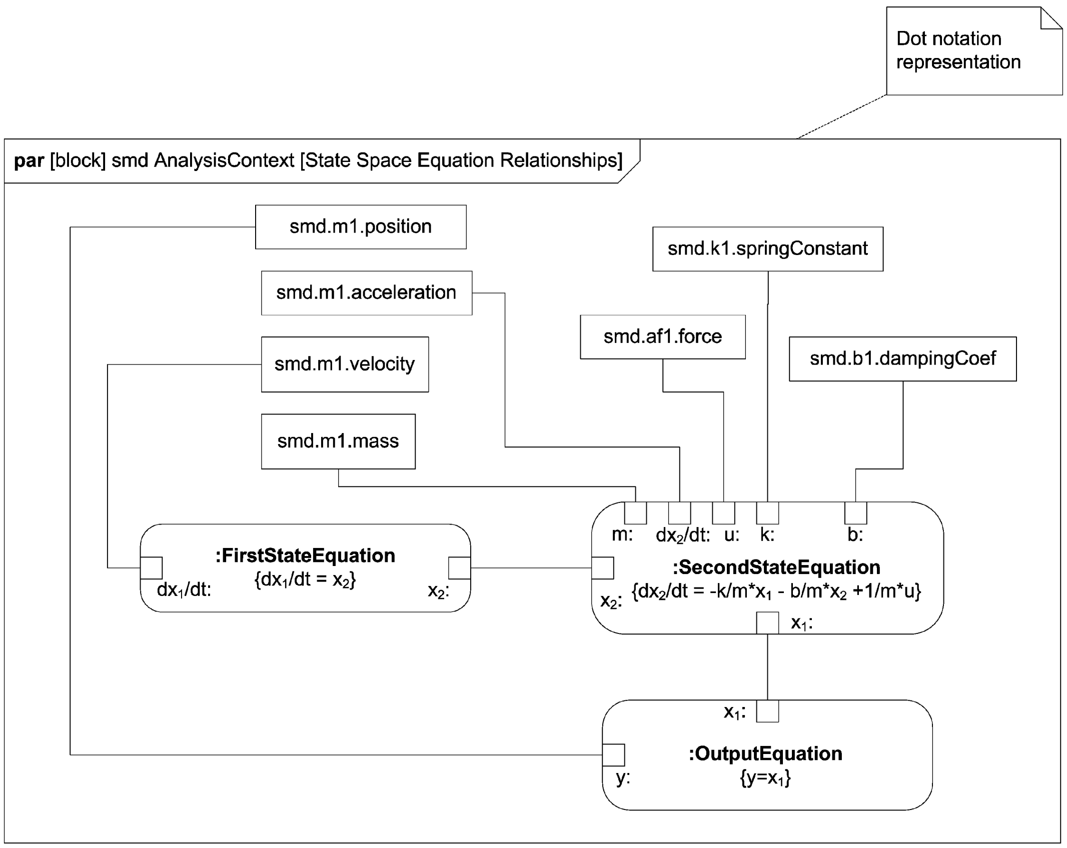


Figure 6. Parametric diagram (**par**) using dot notation showing the state-space equations for a particular SpringMassDashpot system.

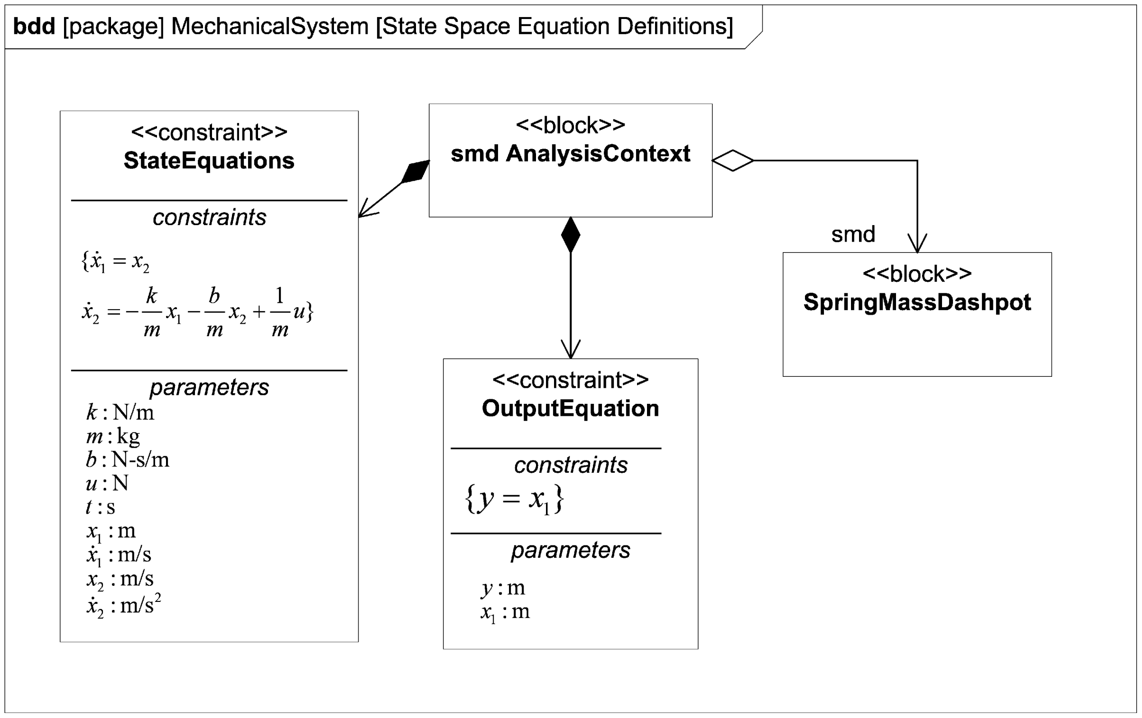


Figure 7. Block definition diagram (**bdd**) for the mechanical system, with two equations in one constraint of the StateEquation.

3. STATE-MACHINE SYSTEMS

3.1. Spelling Checker

State-machine systems are studied in traditional digital logic books such as Katz [1993] and in contemporary object-oriented books such as Blaha and Rumbaugh [2005]. The following is problem 3-11 from Chapman, Bahill, and Wymore [1992]. Design a system for detecting spelling errors. Or, more simply, create a state machine to implement the spelling rule "i before e except after c." If a word violates this rule, your system should stop processing words and produce an error signal. When the human acknowledges the mistake and turns off the error signal, your system should resume processing words. For example, the words *piece* and *receive* are correct so your system should continue processing words. However, *weird* violates this rule, so your system should stop and wait for human action. You may assume that bizarre sequences such as "ceie" will go undetected, and that Professor Lucien Duckstein will not use it. Describe your inputs, outputs, and states. Label your states with meaningful names. Assume the system starts in a reset state. State all assumptions you make. Finally, describe how you will test your system.

3.2. The Use Case Approach

Name: Check Spelling
Iteration: 1.4
Derived from: Problem 3-11 of Chapman, Bahill, and Wymore [1992]
Brief description: The system scans a document and implements the spelling rule "i before e except after c."
Added value: Author gets a more professional document.
Level: Low
Scope: The document being processed
Primary actor: Author
Supporting actor: Document (We are not using a dictionary)
Frequency: Many times per day
Precondition: A document must be open.
Trigger: Author asks the system to check the spelling.
Main Success Scenario:
1a. System scans the document one letter at a time. If it finds a spelling-rule violation, it turns on the error signal and waits for Author to respond.
2. Author resets the system [repeat at step 1].
Alternate Flows:
1b. System reaches the end of the document [exit use case].
Postcondition: The whole document has been checked for violations of this spelling rule.
Specific Requirements
Functional Requirements:
The system shall check a document and find violations of the spelling rule "i before e except after c."
The system shall signal a violation until turned off by the Author.
Nonfunctional Requirements: The system shall ignore numbers, blank spaces, and punctuation.
Author/owner: Terry Bahill
Last changed: January 1, 2007

3.2.1. Define Input Ports, Output Ports, and Their Legal Values

There are two input ports: the character stream and the human reset.

Input Port 1 = {a-z, A-Z}, This port accepts lowercase and uppercase letters, and there is no difference between uppercase and lowercase letters. Numbers, spaces, and punctuation will be ignored. We will label the inputs "c" for the character c, "e" for the character e, "i" for the character i, and "other" for any other character. We are using lowercase letters for the inputs and uppercase letters in the state names.

Input Port 2 = {humanReset, $\overline{\text{HR}}$}, The human either presses the reset button or he does not ($\overline{\text{HR}}$).

There is one output port: the error signal.

Output Port = {OK, error}. Items in braces { } are legal values for that port.

Inputs of state-machine systems should be simple. Typical input ports are switches that can be open or closed. Inputs should not be described as having memory. For example, in this spelling checker you should not describe the *input* as "c received then e received then i received." However, such a description could be the name of a state. Inputs should be named with nouns or noun phrases.

Most systems not only have many inputs, but also have many *kinds* of inputs. Each different kind of input may be designated as an *input port.* There is sometimes a question as to whether a model should have one input with multiple input values or multiple input ports with fewer input values. Inputs that can occur simultaneously must be assigned to different input ports. Inputs that have different values and cannot occur simultaneously can be allowed in the legal value set for one port. *An input port has a set of allowed values each of which is possible individually, but no two can occur simultaneously.* Input ports are interfaces.

Input ports are *not* UML events. An *event* is an occurrence that has a location in time and space. In state machine diagrams, events are the inputs that cause transitions between states. Events should be named with verbs or verb phrases. Typically, the first letter of the first word is lowercase, the first letters of subsequent words are uppercase, and there are no spaces between the words: this is called lower camel case. Events may be composed of Boolean expressions and guard conditions.

In classical state machines, the transitions between states were labeled with the current values for the input, e. g. (c, $\overline{\text{HR}}$), where $\overline{\text{HR}}$ means No humanReset. In UML state machine diagrams, these transitions are labeled as events, e.g., "c received [$\overline{\text{HR}}$]." Things in square brackets [ ] are guard conditions.

In other words, classical state machines track the values of all input ports at all times, whereas UML state machines only show important *changes* of input values.

3.2.2. Assumptions

The list of assumptions is not written at the beginning of the project. It is started at the beginning of design and finished at the end of testing.

- Bizarre sequences such as "ceie" will go undetected.
- Professor Lucien Duckstein will not use the spelling checker.
- The system should ignore numbers, blank spaces, and punctuation.
- The system will not distinguish between upper and lower case letters.
- The Author will not press the reset button unless the error signal is on.
- The human reset input port has priority over the character stream input port.

3.2.3. Define the States

Concurrency and hierarchies will not be used. The following seven states were determined during thoughtful reasoning with much iteration.

Start, /* This is where we start. */
C Received, /* This means the letter c (upper or lower case) was read from the document. */
C Then I, /* This means the letter c was read and immediately after there was an i. */
C Then I Then E, /* This means a c was read, then an i and immediately after an e was read.*/
C Then E, /* This means the letter c was read and immediately after there was an e. */
E Received, /* This means the letter e (upper or lower case) was read from the document. */
E Then I. /* This means the letter e was read and immediately after there was an i. */

3.3. UML/SysML State Machine Diagram

The UML/SysML state machine (**sm**) diagram in Figure 8 nicely describes the system states. This diagram must be accompanied by text such as that given above describing the inputs, outputs and states.

3.3.1. Construction Hints

When creating a state machine model, it usually helps to start with the main success scenario. In this case, success is paradoxically detecting errors. So our two main success scenarios are detecting the sequences ei and cie. So first, we constructed the states Start, E Received, and E Then I and their associated transitions. Then we constructed the branch with the states Start, C Received, C Then I, and C Then I Then E. After this, we added all other possible transitions. While doing this we discovered the need for another state, C Then E. As a final step, we ensured that all states have all of the necessary transition out of them.

In this state machine diagram, it looks like there is only one input port. This results from the assumption that the human would not press the reset button unless the error signal was on; i.e., the input humanReset (HR) would only be activated from the error states: C Then I Then E and E Then I. Without this assumption each arrow would have to have two inputs specified; e.g., the arrow from Start to C Received would be labeled "(c, $\overline{\text{HR}}$)", where $\overline{\text{HR}}$ means Not HR and there would have to be humanReset arrows from states C Received, C Then I, C Then E, and E Received back to Start.

A weakness of the above model is that it does not consider blank spaces or punctuation. Therefore, it would flag this sentence as an error.

Over the last 30 years, we have evaluated thousands of solutions to this problem. They all have strengths and weaknesses. Here is a higher-level solution suggested by a reviewer. "I would define states called 'start', 'scan' and 'error processing'. A scan state would transition to error processing if an error is detected, and certain actions would be performed. The completion of the error processing actions would result in a transition back to the scan state. A reset would take you back to the start state from either the scan or error processing state. There are many sophisticated features of a state machine that could be applied if desired."

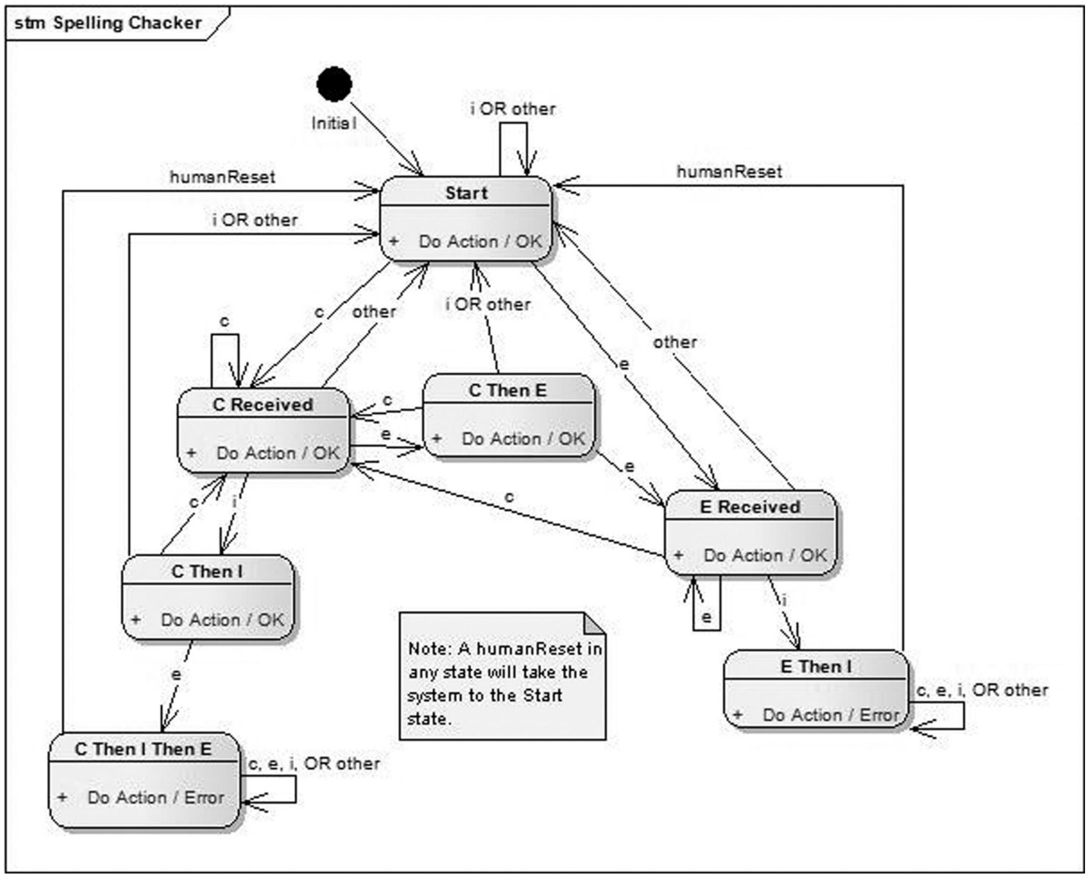


Figure 8. State machine (**sm**) diagram for the spelling checker.

3.4. Other UML and SysML Diagrams

Would any other UML diagrams be useful in describing this system? An activity diagram could be used to describe the algorithm. A sequence diagram would have three objects at the top: the Author, the Document, and the Spelling Checker. The Spelling Checker would go through the document letter by letter and would occasionally send a message to the Author. This does not sound useful, nor do timelines, class diagrams, or deployment diagrams. The SysML extensions to the UML would classify Input Port 1 as a flow port because the characters {a-z, A-Z} flow through this port, Input Port 2 (the human reset) as a standard port, and the Output Port as a standard port.

Our design process starts with the use cases: The requirements and everything else follows. Therefore, the dependencies in Figure 9 show that, for example, the requirement Find Violations *refines* the use case Check Spelling. If, instead, the customer gave us all of the requirements and then we wrote the use cases, then the arrow's direction would show that the use case refines the requirements.

3.5. Wymorian Notation

Here is one possible Wymorian solution. Because this is problem 3-11 of Chapman, Bahill, and Wymore [1992], let us name it Z11.

Z11 = (SZ11, IZ11, OZ11, NZ11, RZ11), where
 SZ11 = {Start, C Received, C Then I, E Received, C Then E, C Then I Then E, E Then I}, /* There is one state variable that can take on one of 7 values. */
 IZ11 = I1Z11 I2Z11, /* There are two input ports. */
  I1Z11 = ALPHABET = {a-z, A-Z}, /* This port accepts lowercase and uppercase letters, and there is no difference between uppercase and lowercase letters. Numbers, spaces and punctuation will be ignored. We will label the inputs "c" for the character c, "e" for the character e, "i" for the character i, and "other" for any other character. We are using lowercase letters for the inputs and uppercase letters

in the state names. DC means we don't care about the value.*/
I2Z11 = {humanReset, $\overline{\text{HR}}$}, /* Either the human presses the reset button or he does not ($\overline{\text{HR}}$). */
/* There is one output port, the error signal. */
OZ11 = {OK, error},
/* Each entry in the next state function is of the form ((present state, (input1, input2)), next state).*/
NZ11 = {((Start, (c, $\overline{\text{HR}}$)), C Received),
((Start, (e, $\overline{\text{HR}}$)), E Received),
((Start, (i, $\overline{\text{HR}}$)), Start),
((Start, (other, $\overline{\text{HR}}$)), Start),
((Start, (DC, humanReset)), Start),
((C Received, (c, $\overline{\text{HR}}$)), C Received),
((C Received, (e, $\overline{\text{HR}}$)), C Then E),
((C Received, (i, $\overline{\text{HR}}$)), C Then I),
((C Received, (other, $\overline{\text{HR}}$)), Start),
((C Received, (DC, humanReset)), Start),
((C Then I, (c, $\overline{\text{HR}}$)), C Received),
((C Then I, (e,$\overline{\text{HR}}$)), C Then I Then E),
((C Then I, (i, $\overline{\text{HR}}$)), Start),
((C Then I, (other, $\overline{\text{HR}}$)), Start),
((C Then I, (DC, humanReset)), Start),
((C Then I Then E, (c, $\overline{\text{HR}}$)), C Then I Then E),
((C Then I Then E, (e, $\overline{\text{HR}}$)), C Then I Then E),
((C Then I Then E, (i, $\overline{\text{HR}}$)), C Then I Then E),
((C Then I Then E, (other, $\overline{\text{HR}}$)), C Then I Then E),
((C Then I Then E, (DC, humanReset)), Start),
((C Then E, (c, $\overline{\text{HR}}$)), C Received),
((C Then E, (e, $\overline{\text{HR}}$)), E Received),
((C Then E, (i, $\overline{\text{HR}}$)), Start),
((C Then E, (other, $\overline{\text{HR}}$)), Start),
((C Then E, (DC, humanReset)), Start),
((E Received, (c, $\overline{\text{HR}}$)), C Received),
((E Received, (e, $\overline{\text{HR}}$)), E Received),
((E Received, (i, $\overline{\text{HR}}$)), E Then I),
((E Received, (other, $\overline{\text{HR}}$)), Start)},
((E Received, (DC, humanReset)), Start),
((E Then I, (c, $\overline{\text{HR}}$)), E Then I),
((E Then I, (e, $\overline{\text{HR}}$)), E Then I),
((E Then I, (i, $\overline{\text{HR}}$)), E Then I),
((E Then I, (other, $\overline{\text{HR}}$)), E Then I),
((E Then I, (DC, humanReset)), Start)},
/* Each entry in the readout function is of the form (present state, output).*/
RZ11 = {(Start, OK),
(C Received, OK),
(C Then I, OK),
(C Then I Then E, error),
(C Then E, OK),
(E Received, OK),
(E Then I, error)}.

Notice that in Wymorian modeling a system must have both an input port and an output port.

3.5.1. Alternative Wymorian Solution

It is often convenient to manipulate the values of the inputs, states, next states, and outputs. When this is done, we will use p1 and p2 for particular values of the first and second inputs, x for a particular value of the state, y for a value of the next state, and q for a value of the output. Using this notation, we can write the following NZ11, which is more eloquent than the NZ11 above.

NZ11 = {((x, p), y): x ∈ SZ11; p I1Z11; y∈ SZ11;
{if p2 = humanReset, then y = Start}
∪ {((Start, (p1, $\overline{\text{HR}}$)), y):
if p1 ∉ {c, e}, then y = Start,
else if p1 = c, then y = C Received,
else y = E Received}
∪ {((C Received, (p1, $\overline{\text{HR}}$), y):
if p1 = c, then y = C Received,
else if p1 = i, then y = C Then I,
else if p1 = e, then y = C Then E,
else y = Start}
∪ {((C Then I, (p1, $\overline{\text{HR}}$), y):
if p1 = c, then y = C Received,
else if p1 = e, then y = C Then I Then E,
else y = Start}
∪ {etc.}.

where ∪ represents the union of sets.

3.6. Testing the Spelling Checker

The easiest way to test the system might be to build one and run it on a list of words that you know are correct, such as the dictionary of a computer spell program, and also on a list of known incorrect words. For a paper and pencil test of your paper and pencil state machine, you could use the following words.

Correct: field, percent, believe, tiniest, piece, decide, receive
Incorrect: handkercheif, yeild
Exceptions that proof the rule: their, height, being, sufficiently, weird

These exceptions should also be flagged as errors.

Do we have enough words? Do we have the right words? Have we exercised every arrow on the state diagram? Have we exercised some paths many times? Can you suggest a more systematic way of testing this system?

We could use a dozen words from the dictionary that contain the several correct variations of i's, e's, and c's

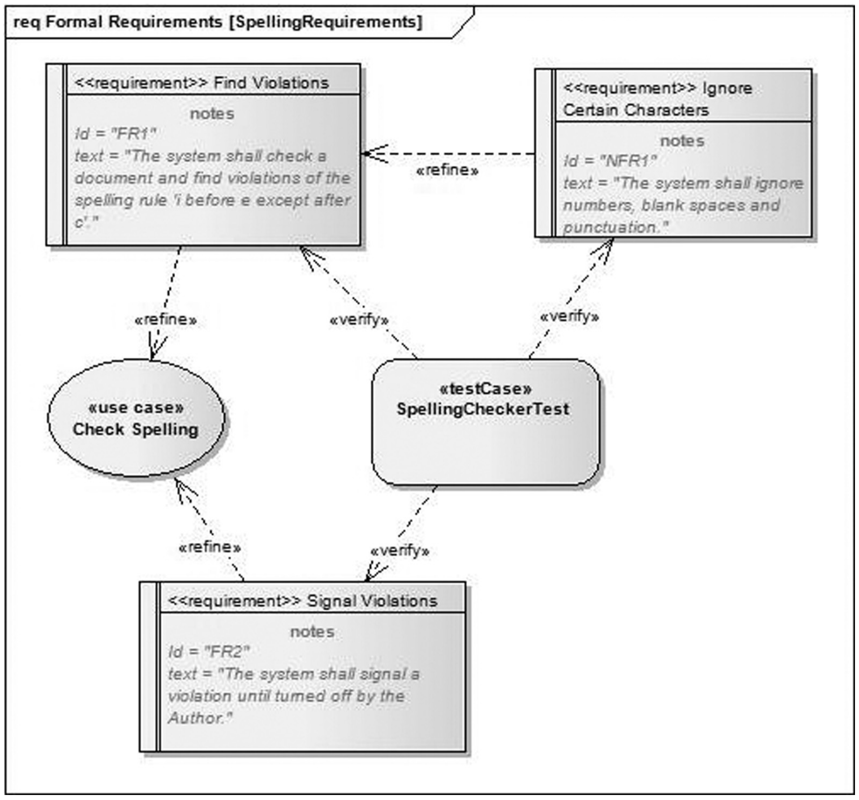

req Formal Requirements [SpellingRequirements]
<<requirement>> Find Violations
notes
Id = "FR1"
text = "The system shall check a document and find violations of the spelling rule 'i before e except after c'."
<<requirement>> Ignore Certain Characters
notes
Id = "NFR1"
text = "The system shall ignore numbers, blank spaces and punctuation."
«refine»
«refine»
«verify»
«verify»
«use case» Check Spelling
«testCase» SpellingCheckerTest
«refine»
«verify»
<<requirement>> Signal Violations
notes
Id = "FR2"
text = "The system shall signal a violation until turned off by the Author."

Figure 9. Requirements diagram for the Spelling Checker.

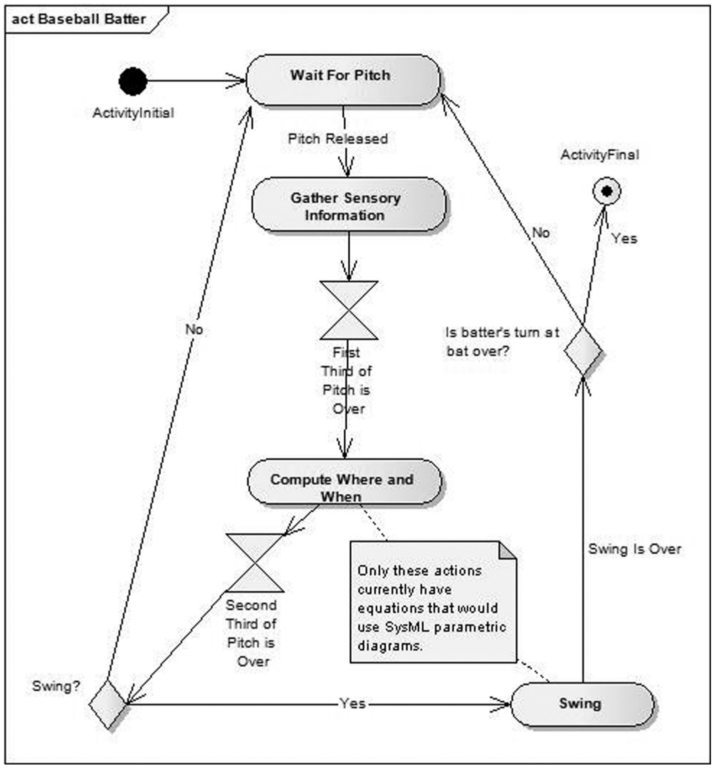

act Baseball Batter
ActivityInitial
Wait For Pitch
Pitch Released
Gather Sensory Information
First Third of Pitch is Over
Compute Where and When
Second Third of Pitch is Over
Only these actions currently have equations that would use SysML parametric diagrams.
Swing?
No
Yes
Swing
Swing Is Over
Is batter's turn at bat over?
No
Yes
ActivityFinal

Figure 10. Activity diagram (**act**) for the batter in a baseball game.

and make sure that they were accepted as correct. Then transpose all of the ie's and ei's and make sure that they were then flagged as incorrect.

If we actually built our system, we could use other techniques to help validate it. For example, if we built our spelling checker and tested it on 100,000 words, we could record the number of times each state was entered. If we found a state that was never entered, we might suggest that it was a mistake. Likewise, if we found a state that was entered 100,000 times, we might suggest that it was either a start-state or a mistake. States that are entered for all or no test trajectories are always suspicious.

We conclude this problem with the following anonymous poem.

My Spelling Checker
I have a spelling checker,
It came with my PC.
It plainly marques four my revue,
Miss steaks eye can knot sea.
I've run this poem threw it,
I'm sure your pleased too no;
Its letter perfect in it's weigh,
My check her tolled me sew.

4. STATE-EQUATION VERSUS STATE-MACHINE SYSTEMS

4.1. Comparison

Section 2 of this paper presented state equation models. Section 3 presented state-machine models. Table II gives a general comparison. This table is not rigorous or mathematical; instead, it is informal and heuristic.

Next, we want to show a system that is modeled best with a combination of state equations *and* state machines. However, to illustrate different modeling possibilities, we will model the exact same thing with both an activity diagram and a state machine diagram.

4.2. An Equivocal Problem Domain

4.2.1. The Activity Diagram

Now we want to discuss a model for the batter in a game of baseball. For the batter, the pitch can be divided into thirds. During the first third of the pitch, the batter gathers sensory information. During the second third, he or she must compute *where* and *when* the ball will make contact with the bat and then decide whether to swing, duck, or take the pitch. During the final third of the pitch, the batter is swinging the bat and can do almost nothing to alter its trajectory. The activity diagram of Figure 10 is very useful for describing the batter's actions during the pitch. Computing where and when has been modeled with equations [Bahill and Karnavas, 1993] and the swing of the bat has also been modeled with equations [Watts and Bahill, 2000]. Therefore, SysML block definition and parametric diagrams could be used to describe these equations. These would be the only actions in this diagram that are described this way. We do not have equations that describe wait for the pitch or gather sensory information.

In this activity diagram for the batter, the hourglass shape is a Timing signal. In the Enterprise Architecture (EA) tool, this is a specialization of the Accept signal.

It seems that a UML sequence diagram and a use case would also be useful in describing the behavior of the batter. The SysML block definition and parametric diagrams would be good for describing the dynamics of the batter.

The activity diagram has evolved. In 1967, Jim Long developed a technique for representing system logic that incorporated functions, inputs, outputs, and control in a single hierarchical set of diagrams [Long, 2002].

Table II. Comparison of State-Equation Systems and State-Machine Systems

State-equation systems	State-machine systems
Focus on physical variables	Focus on logical variables
Are modeled with differential or difference equations	Are modeled with finite state machines
Can be continuous or discrete	Discrete only
They use feedback to improve performance and therefore have potential stability problems.	They do not use feedback to improve performance and consequently stability is only an esoteric (or an implementation) concern.
Typical state-equation systems include analog computers as well as electrical, mechanical, hydraulic, pneumatic, thermal, economic, chemical and biological systems.	Anything containing a digital processor can be modeled as a state-machine system.
The main focus is dynamics.	The main focus is behavior.
Have no useful UML diagrams	Have several useful UML behavior diagrams
Have SysML block definition and parametric diagrams	SysML provides additional modeling facets

This developed into the extended functional flow block diagram, which, with the addition of the colored Petri-net token concept, was the basis for the SysML activity diagram [Bock, 2006].

A good model is a simplified representation of one aspect of a real system: models are successful because they do not consider all the complexity of the real system. As with all successful models, the example of Figure 10 only models one aspect of the batter's behavior. The actual batter is doing many other tasks in parallel [Bahill and LaRitz, 1984; McHugh and Bahill, 1985; Bahill and Karnavas, 1992, 1993; Watts and Bahill, 2000; Bahill and Baldwin, 2003, 2004, 2007; Bahill, Baldwin, and Venkateswaran, 2005; Bahill, Botta, and Daniels, 2006; Baldwin, Bahill, and Nathan, 2007]. Including many tasks in the same model is detrimental to understanding [Mellor et al., 2004].

4.2.2. The State Machine Diagram

Now we want to discuss another model for the batter in a game of baseball. For the batter, the pitch can be divided into thirds. During the first third of the pitch, the batter gathers sensory information. In this state, the batter must block out all distractions and concentrate on the ball. During the second third, he or she must compute *where* and *when* the ball will make contact with the bat and then decide whether to swing, duck or take the pitch. During the final third of the pitch, the batter is swinging the bat and can do almost nothing to alter its trajectory. The state machine diagram of Figure 11 is very useful for describing the batter's behavior during the pitch. Computing where and when the ball will cross the plate has been modeled with equations [Bahill and Karnavas, 1993] and the swing of the bat has also been modeled with equations [Watts and Bahill, 2000]. SysML block definition and parametric diagrams could be used to describe these equations. These would be the only states in this diagram that are described this way. We do not have equations that describe waiting for the pitch or gathering sensory information.

It seems that a UML activity diagram, sequence diagram and a use case would also be useful in describing the behavior of the batter. The SysML block definition and parametric diagrams would be good for describing the dynamics of the batter.

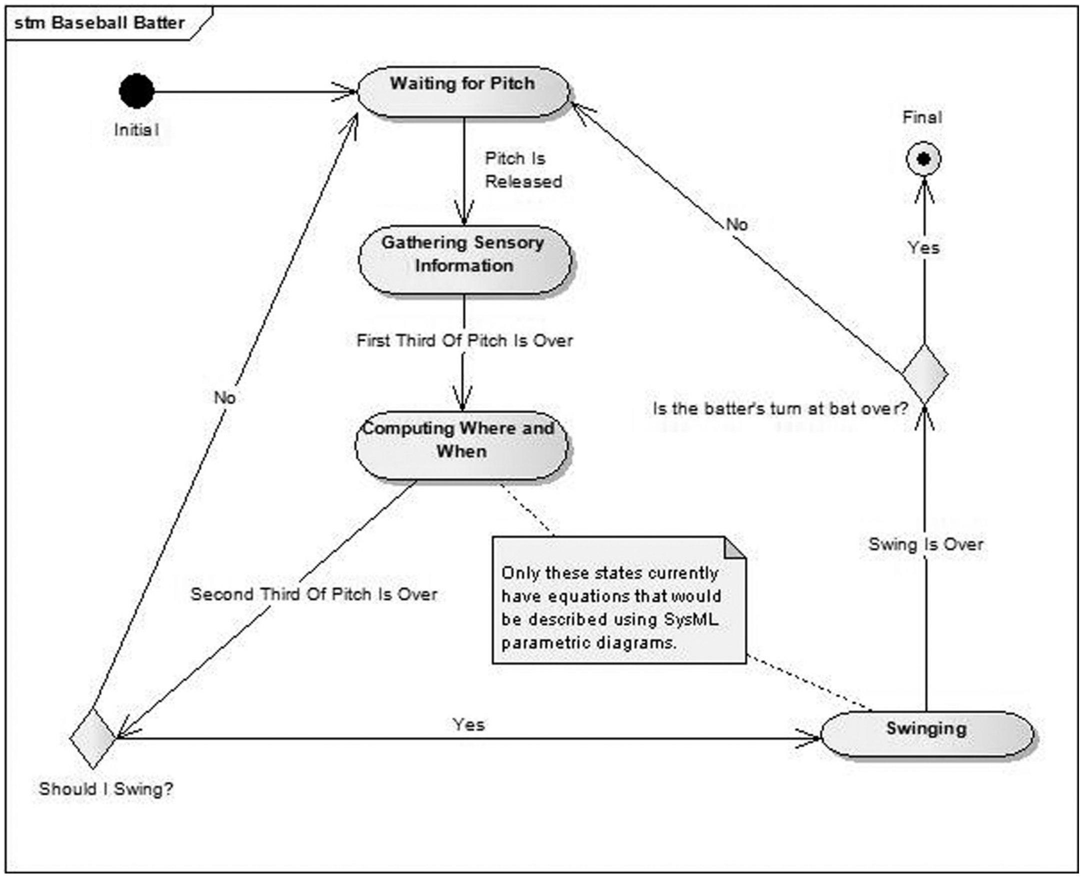

stm Baseball Batter
Initial
Waiting for Pitch
Pitch Is Released
Gathering Sensory Information
First Third Of Pitch Is Over
Computing Where and When
Second Third Of Pitch Is Over
Should I Swing?
No
Yes
Swinging
Swing Is Over
Is the batter's turn at bat over?
No
Yes
Final
Only these states currently have equations that would be described using SysML parametric diagrams.

Figure 11. State machine diagram (**sm**) for the batter in a baseball game.

4.2.3. Which is the Best Model, the Activity or the State Machine Diagram?

Maybe we could answer this question by explaining if waiting for the pitch, gathering sensory information, computing where and when, and swinging the bat are states or actions. But there is no clear answer to this dilemma. If we were to expand the model upward to include learning how to swing the bat and how to compute where and when the ball will cross the plate, then we might want to use the state machine diagram. However, if we were to expand the model downward to include how the batter predicts time to contact and how he uses the spin on the ball to predict the spin-induced deflection of the ball, then we might want to use the activity diagram. What about the transitions? Transitions between states are triggered by events. Pitch Is Released certainly sounds like an event, but First Third Of Pitch Is Over does not. The final thing that might influence our choice is that the batter is not likely to make decisions in parallel; therefore, the activity diagram loses its forte. In conclusion, in many cases it will not be clear whether an activity diagram or a state machine diagram would be best. Creating rules for making such a decision is clearly outside the scope of this paper.

5. MIXED MODELS

For some systems, it might be necessary to use several modeling methods. To understand neurological systems, we make models at different levels of abstraction; for example, we model neurons, neural networks and the resulting behavior. The Hodgkin-Huxley equations for the voltage potential across a neural membrane use differential equations, whereas most models for artificial neural networks use difference equations; and, then, models of the behavior of neurological networks usually use state machines. Combining all three of these levels in the same model is difficult, and few people try. So we do not find such mixed models for the brain.

However, in designing complex systems, such as an air traffic control system or a communications network, it might be necessary to use all of these methods together. The design might start with the use cases that model the required system behavior and capture the requirements. Then by thinking about concepts in the problem domain, we identify the classes. Sequence diagrams then show the interactions between these classes, and they help us to find the operations (functions) of the classes. Block definition diagrams show system structure as components along with their properties, constraints, parameters, and relationships. State machine diagrams describe event-based behavior in a part of the life cycle of a class or a block. State-equations are used to describe the dynamics of the system. Parametric diagrams show how the parameters from each of the equations can be bound together to create a complex network of equations. Requirements diagrams show the requirements and their relationships to each other, to use cases and to test cases. So, although we described each of these methods separately in this paper, in the design of complex systems, it may be necessary to combine many of these methods in the same model or design.

6. CONCLUSIONS

Model-based System Engineering is a relatively new concept in system design. It requires good consistent rigorous simulatable models. A good model is a simplified representation of *one* aspect of a real system: Models are successful *because* they do not simultaneously consider all the complexity of the real system. Simple models are easier to understand and solve than models that are more complex. Each problem domain requires an appropriate modeling method. This paper presented state-equation and state-machine problem domains. Most importantly, all of the examples were at the same level of abstraction. It presented continuous system and discrete system problem domains. It used the following methods for modeling systems: the state-space approach of Linear Systems Theory, set-theoretic notation, block diagrams, use cases, UML diagrams, and SysML diagrams. The main conclusion of this paper is that Model-based System Engineering requires correct models that are appropriate to the particular problem domain. The systems engineer must first carefully study the problem in order to understand which modeling method will be most suitable. One method will not work for all problems. For some problems, two methods may work equally well, but usually there is a best method. The method must be chosen for the particular problem.

ACKNOWLEDGMENT

Sandy Friedenthal, a Principal System Engineer at Lockheed Martin and co-organizer of the SysML, made major contributions to the SpringMassDashpot SysML section. We thank Sparx Systems for providing a site license for Enterprise Architect.

REFERENCES

A.T. Bahill and D.G. Baldwin, The vertical illusions of batters, Baseball Res J 32 (2003), 26–30.

A.T. Bahill and D.G. Baldwin, "The rising fastball and other perceptual illusions of batters," Biomedical engineering principles in sports, George Hung and Jani Pallis (Editors), Springer, New York, 2004, pp. 257–287.

A.T. Bahill and D.G. Baldwin, Describing baseball pitch movement with right-hand rules, Comput Biol Med 37 (2007), 1001–1008.

A.T. Bahill and W.J. Karnavas, "Bat Selector," U. S. Pat. 5,118,102, June 2, 1992.

A.T. Bahill and W.J. Karnavas, The perceptual illusion of baseball's rising fastball and breaking curve ball, J Exper Psychol: Hum Perception Performance 19 (1993), 3–14.

A.T. Bahill and T. LaRitz, Why can't batters keep their eyes on the ball, Amer Scientist 72 (May–June 1984), 249–253.

A.T. Bahill, D.G. Baldwin, and J. Venkateswaran, Predicting a baseball's path, Amer Scientist 93(3) (May–June 2005), 218–225.

A.T. Bahill, R. Botta, and J. Daniels, The Zachman framework populated with baseball models, J Enterprise Architecture 2(4) (2006), 50–68.

A.T. Bahill, F. Szidarovszky, R. Botta, and E.D. Smith, Valid models require defined levels, Int J Gen Syst 37(5) (2008), 553–571.

D.G. Baldwin, A.T. Bahill, and A. Nathan, Nickel and dime pitches, Baseball Res J 35 (2007), 25–29.

M. Blaha and J. Rumbaugh, Object-oriented modeling and design with UML, 2nd edition, Pearson, Prentice Hall, Englewood Cliffs, NJ, 2005.

C. Bock, SysML and UML 2 support for activity modeling, Syst Eng 9(2) (2006), 160–186.

R. Botta, Z. Bahill, and A.T. Bahill, When are observable states necessary? Syst Eng 9(3) (2006), 228–240.

D.M. Buede, The engineering design of systems: Models and methods, Wiley, New York, 2000.

W.L. Chapman, A.T. Bahill, and W. Wymore, Engineering modeling and design, CRC Press, Boca Raton, FL, 1992.

M. Fowler, UML distilled: A brief guide to the standard object modeling language, 3rd edition, Addison-Wesley, Reading, MA, 2004.

S. Friedenthal, A. Moore, and F. Steiner, OMG Systems Modeling Language, (OMG SysML™), INCOSE Tutorial, June 25, 2007, http://www.omgsysml.org/INCOSE-2007-OMG-SysML-Tutorial.pdf, accessed March 2008.

R.H. Katz, Contemporary logic design, Benjamin/Cummings, Redwood City, CA, 1993.

J. Long, Relationships between common graphical representations in systems engineering, http://www.vitechcorp.com/whitepapers/files/200701031634430.CommonGraphicalRepresentations_2002.pdf, accessed March 7, 2008.

D.E. McHugh and A.T. Bahill, Learning to track predictable target waveforms without a time delay, Investigative Ophthalmol Vis Sci 26 (1985), 932–937.

S.J. Mellor, K. Scott, A. Uhl, and D. Weise, MDA distilled: Principles of model driven architecture, Addison-Wesley, Reading, MA, 2004.

I. Newton, The philosophiae naturalis principia mathematica, 1687.

K. Ogata, System dynamics, 4th edition, Prentice Hall, Englewood Cliffs, NJ, 2004.

OMG SysML, OMG Systems Modeling Language, The Official OMG SysML site, http://www.omgsysml.org/, 2007.

OMG UML, Unified Modeling Language, UML Resource Page, http://www.uml.org/, 2007.

J.W. Rosenblit, A conceptual basis for model-based system design, PhD dissertation in Computer Science, Wayne State University, Detroit, MI, 1985 (published by University Microfilms International, Ann Arbor, MI).

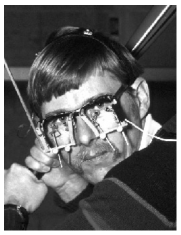
SysML—Open Source Specification Project, http://www.sysml.org/, 2007.

F. Szidarovszky and A.T. Bahill, Linear systems theory, 2nd edition, CRC Press, Boca Raton, FL, 1998.

A. Turing, On computable numbers with an application to the Entscheidungsproblem, Proc London Math Soc XLII (1936), 239–265 [correction: Proc London Math Soc XLIII (1937), 544–546].

R.G. Watts and A.T. Bahill, Keep your eye on the ball: Curve balls, knuckleballs and fallacies of baseball, Freeman, New York, 2000.

A.W. Wymore, Model-based systems engineering, CRC Press, Boca Raton, FL, 1993.



A. Terry Bahill is a Professor of Systems Engineering at the University of Arizona in Tucson. He received his Ph.D. in electrical engineering and computer science from the University of California, Berkeley, in 1975. His research interests are in the fields of system design, systems engineering, modeling physiological systems, eye-hand-head coordination, human decision-making, and systems engineering theory. He has tried to make the public appreciate engineering research by applying his scientific findings to the sport of baseball. Bahill has worked with BAE Systems in San Diego, Hughes Missile Systems in Tucson, Sandia Laboratories in Albuquerque, Lockheed Martin in Eagan MN, Boeing in Kent WA, Idaho National Engineering and Environmental Laboratory in Idaho Falls, and Raytheon Missile Systems in Tucson. For these companies he presented seminars on Systems Engineering, worked on system development teams, and helped them describe their Systems Engineering Process. He holds a U.S. patent for the Bat Chooser, a system that computes the Ideal Bat Weight for individual baseball and softball batters. He received the Sandia National Laboratories Gold President's Quality Award. He is a Fellow of the Institute of Electrical and Electronics Engineers (IEEE), of Raytheon and of the International Council on Systems Engineering (INCOSE). He is the Founding Chair Emeritus of the INCOSE Fellows Selection Committee. His picture is in the Baseball Hall of Fame's exhibition "Baseball as America." You can view this picture at http://www.sie.arizona.edu/sysengr/.

Ferenc Szidarovszky has been a professor of Systems and Industrial Engineering at the University of Arizona in Tucson since 1987. He was born in Budapest, Hungary and received his B.S., M.S., and Ph.D. in numerical techniques from the Eötvös University of Sciences in Budapest. He received a second Ph.D. in economics from the Budapest University of Economic Sciences in 1977. He was an assistant and an associate professor in the Department of Numerical Analysis and Computer Sciences of the Eötvös University of Sciences. He served as the Acting Head of the Department of Mathematics and Computer Sciences of the University of Horticulture and Food Industry. He was a professor with the Institute of Mathematics and Computer Sciences of the Budapest University of Economic Sciences. He is vice president of NOAH, a company consulting on applications of neural networks to natural resource management. He also presents courses world wide on game theory, multiple criteria decision making, and conflict resolution.