Regular Paper

When Are Observable States Necessary?

Rick Botta,[1] Zach Bahill,[2] and Terry Bahill[3,*]

[1]*BAE Systems, 10920 Technology Place, San Diego, CA 92127*

[2]*Boeing Integrated Defense Systems, Kent, WA*

[3]*Systems and Industrial Engineering, University of Arizona, Tucson, AZ 85721-0020*

Received 17 August 2005; Revised 7 November 2005; Accepted 3 March 2006, after one or more revisions
Published online in Wiley InterScience (www.interscience.wiley.com).
DOI 10.1002/sys.20053

ABSTRACT

In order to use commercial off-the-shelf (COTS) products, the engineer must be able to prove that the COTS product is equivalent to the specified design. In most cases, this requires observable states, which are usually not available, because the supplier may not know or may not want to disclose the internal states of the system. This paper first presents the following reasons for proving system equivalence: to reuse existing systems, to upgrade systems, to use COTS products, to replicate failures, to verify that a physical system conforms to its design, and to test evolving systems. Next, the paper presents the following techniques that have been used in lieu of proving system equivalence: Create multiple reset (or test) states and prove I/O equivalence with respect to all initial state pairs, implement built-in self-tests, use regression testing, define pre and post conditions, only use COTS products in places where you do not care about observable states, put a wrapper around COTS products, record the mode the system was in when the event of interest occurred, abstract the code into a state machine, build an observer to estimate the system states, and add extra outputs so that the states can be identified by examining the outputs. Finally, this paper gives examples where states are necessary and unnecessary in modeling systems. © 2006 Wiley Periodicals, Inc. Syst Eng 5: 228–240, 2006

Key words: state machines; system equivalence; reuse; testing; verification; COTS

Contract grant sponsor: AFOSR/MURI F49620-03-1-0377.

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

1. THE DECISION OF MAKE VERSUS REUSE VERSUS BUY

Are you going to make, reuse or buy lunch today? You could go to a restaurant and buy a hamburger. Or you could reuse your leftover pizza (if it's not moldy). Or you could decide to make lunch. In which case, you could go to the grocery store and buy hamburger, lettuce, tomatoes, cheese, buns, and mayonnaise. However, rather than buying the mayonnaise, you could reuse the jar you have in the refrigerator (after checking the expiration date) or you could decide to make the mayonnaise after buying salt, vinegar, lemon juice, oil, and eggs. Rather than buying the eggs, you could buy a chicken and … The point is, almost all make-reuse-buy decisions end up with the decision to buy at some level.

MIL-STD 499 said the way to design a system was to do functional decomposition. Students used to query, "When do you stop decomposing?" We used to answer, "When you get a function small enough to be designed by a team of people." Now we usually answer, "When you get a function that can be implemented by a commercial off-the-shelf product."

So it is now apparent that in designing systems we are going to buy commercial off-the-shelf (COTS) products. There are lots of books, papers, and Web sites that give advice about using COTS products. Hundreds of papers were used in detail when we developed the COTS-Based Engineering package, notably those from the Software Engineering Institute [2005], the Systems and Software Consortium [2005], the USC Center for Software Engineering [2005], and the NSF Center for Empirically Based Software Engineering [2005]. However, in these papers we found no suggestions for proving equivalence of COTS products.

This paper is about proving equivalence of systems using state-based techniques. This has been of interest to the digital-system design community for the last half of a century. Because they had control of their designs, they could and did create observable states. But COTS products seldom provide observable states and therefore the desire to use COTS products creates a problem that raises this issue to new prominence.

The paper starts with the premise that many systems are state-based and that proving equivalence of state-based systems cannot be done using only input/output testing. Then it shows six reasons why engineers would want to prove system equivalence. This is the first part of the answer to the question "When are observable states necessary?" The paper follows with ten techniques that engineers use in lieu of complete state-based testing. Next, it defines the difference between memoryless and dynamic systems and provides many examples of each. It shows that equivalence of dynamic systems can only be proven using state-based tests, whereas equivalence of memoryless systems can be proven using only input/output testing. This is the second part of the answer to the question, "When are observable states necessary?"

2. PROVING SYSTEM EQUIVALENCE

The desire to use commercial off-the-shelf products forces the question, "How can we prove that two systems are equivalent?" We will now show six general examples where it is necessary to prove system equivalence. First, suppose a system called Z1 was designed to perform task-1. Next, suppose task-2 needs to be performed. A new system called Z2 could be designed, or perhaps Z1 could be reused. In many cases, it would be a lot cheaper to reuse Z1. For simplicity assume task-2 is a subset of task-1; for example, task-1 could be spelling and grammar checking and task-2 might be only spelling checking. A necessary, but *not* sufficient, condition for using Z1 for task-2 is that the I/O behavior of Z1 and Z2 be identical for task-2. For example, the I/O behavior of Z1 and Z2 would have to be identical when checking spelling. However, identical I/O behavior when checking spelling is *not* a *sufficient* condition for proving equivalence, because the grammar-checking module of Z1 (if it were ever invoked) could have mistakes, trap doors, or Easter eggs that could destroy the spelling checker.

Second, suppose that we have a large complex system that has been working well for several years. But the hardware is getting old and expensive to maintain. Will our application still work if we upgrade from hardware-1 to hardware-2? Or perhaps our software vendor has come out with a new version. Other sites have upgraded, but we have not, so we are losing compatibility. Will our application still work if we upgrade from software-1 to software-2? A necessary but *not* sufficient condition for success of the upgrade is that the input/output behavior of the application be the same on the old system as it is on the new system. However, identical I/O behavior of the application is not a sufficient condition for proving equivalence, because, for example, a portion of hardware-2 might have a switch that disables the functions of hardware-1 if the system gets into a certain undefined state.

Third, suppose we make custom systems for our customer. They work very well, but they are expensive. Now our customer wants us to design a new system, but to keep the cost down, he wants us to use as many commercial off-the-shelf products as possible. We have a design, Z1, that satisfies the customer's needed functionality. But we want to know if a COTS system, Z2, will also satisfy this functionality.

A very common technique for describing the desired behavior of a system is to describe acceptable input/output trajectories (or strings) for the system. Such descriptions of input and output behaviors as functions of time are variously called trajectories, strings, behavioral scenarios, use cases, flows, threads, operational scenarios, logistics, functionality, test vectors, sequence diagrams, or interaction diagrams. When using such techniques the following question often arises: "How do you know when you have enough trajectories?" The answer seems to be *never*. Because merely looking at input/output behavior can never guarantee correct system behavior: we must be able to observe the system's state [Wymore and Bahill, 2000].[1] The state of the system contains all of the information needed to calculate responses to present and future inputs without reference to the past history of inputs and outputs.

Fourth, often field failures reported in deficiency reports cannot be replicated by maintenance or support staff. Sometimes these failures are written off as "pilot error." But, more likely, these problems are a result of not being able to replicate the exact state of the system when the anomaly occurred. A common fix-it technique is to reboot the system. This, of course, removes information about the state the system was in at the time of failure. Having observable states would ameliorate this problem of replicating failures.

Fifth, system verification often requires observable states. Suppose Engineering designs a system, then Manufacturing builds a physical system. Could an engineer prove that the physical system implements the design? If the states were observable, then the engineer could construct an input trajectory (scenario) that exercised all state transitions (changes from one state to another), apply this input trajectory to both systems and compare the resulting state trajectories. If they were identical, then the systems would be equivalent. However, if the system states were not observable, then the only thing the engineer could work with is the input/output behavior of the systems. Wymore and Bahill [2000] proved that identical input/output behavior is not sufficient for proving equivalence of two state-based systems.

Suppose an engineer tried to implement the design Z3 in, say, TTL circuitry. How could he or she test this hardware to prove that it did indeed implement the design? If the states were observable, then the engineer could construct an input trajectory (string) (or a set of input trajectories) that exercised all state transitions, apply this input trajectory to Z3 and to the TTL circuit, and compare the resulting state trajectories. If they were identical, then the systems would be equivalent. However, what if all of the states were not observable? Well, then the engineer could define some equivalent states of the design and of the physical system and then prove I/O equivalence with respect to each initial state pair [Wymore and Bahill, 2000]. Finally, what if you cannot put the system into all possible initial states? Then you cannot prove system equivalence!

Sixth, assume we are using an evolutionary acquisition life cycle model [DoD, 2003]. We build a system and deliver it to our customer. Then, in the next spiral, we add requirements, functionality, and money and deliver an improved system. How do we test the second system? We could treat it as a new system and design tests from the bottom up, but it would be more efficient to reuse previous tests. Therefore, we use the test vectors and test procedures of the first system and add new tests for the new functionality. If the original tests were state-based, this technique will work. If they were merely input/output tests, then it will not.

We have just mentioned six reasons why engineers might want to prove that two systems are equivalent: (1) to reuse existing systems, (2) to upgrade systems, (3) to use commercial off-the-shelf (COTS) products, (4) to replicate failures, (5) to verify that a physical system conforms to its design, and (6) to test evolving systems. Of course, there are many more reasons. One of the best ways to prove system equivalence is to design an input trajectory (or perhaps a set of input trajectories) that exercises every possible state transition, apply it to both systems, and ensure that the two state trajectories are identical. However, this might be impossible, because the states are not observable: or it might be too expensive, due to the large number of states.

Some definitions of system equivalence. If equivalent systems are started in equivalent states, then they produce the same state trajectories for all input trajectories. Equivalent systems have the same input/output behavior for all input trajectories and all pairs of possible starting states. Equivalence is not based on satisfying the same requirements, because requirements are seldom precise enough to define all system states. Equivalence is not based on having the same design, because most designs can have multiple implementations. Mathematically, the equivalence relationship is reflective, symmetric, and transitive. If Z1, Z2, and Z3 are systems, then Z1 is equivalent to Z1 (reflexivity). If Z1 is equivalent to Z2, then Z2 is equivalent to Z1 (symmetry). If Z1 is equivalent to Z2 and Z2 is equiva-

[1] For specific systems where the outputs were created to provide exact state information (called state readout [Wymore, 1993]) and the outputs also provide initial state information, then obviously the state trajectory could be computed from the output trajectory and input/output behavior could be used to prove equivalence.

lent to Z3, then Z1 is equivalent to Z3 (transitivity). The most popular techniques for minimizing systems use equivalence classes [Katz, 1994: 452–460; Hill and Peterson, 1981: 282–300].

Here are some techniques that have been used in lieu of comprehensive state-based testing. (1) Create multiple reset (or test) states and prove I/O equivalence with respect to all initial reset-state pairs. These reset states must be precise. Each system must go into a well-defined reset state when ever the reset signal is received no matter what it was doing or what state it was in. Ctrl-C guaranteed termination of DOS programs. Ctrl-Alt-Delete put the system into a unique reset state for early Windows systems. Now, turning the power off for at least one minute usually produces a reset state for modern Windows systems. Unix and Linux systems are more deterministic. (2) Implement built-in self-tests. The built-in self-tests should verify that the systems satisfy certain requirements. If the same requirements are being verified for both systems, then partial equivalence is being demonstrated. (3) Use regression testing. Create a test suite, a tool that gives an environment to automatically run test cases at specified intervals and report regressions. (4) Use the pre- and post-condition slots of the use case template [Bahill, 2006]. These can be used to create state machine diagrams and subsequent equivalence tests. (5) Only use COTS products in places where you do not care about observable states. For example, in an interface that passes data from a high security system to a low security system, as long as it passes the needed information, we do not care what happens internally. (6) Build a wrapper around the COTS product. Build a state-based interface to go between a COTS product and your system. Then apply input trajectories to the first COTS product, allow its outputs to affect the states of the interface, and record the state trajectories of the interface. Then do the same for the second COTS product. Hopefully the state trajectories will be similar. (7) Record the *mode* the system was in when the event of interest (perhaps a failure) occurred. The state of an airplane would include speed, position, altitude, weight, barometric pressure, humidity, etc. But the mode would be much simpler: loading, taxing, taking off, climbing, cruising and landing. The UML captures this concept of modes by using a hierarchy of superstates and substates. (8) Provide an interactive facility for abstracting software programs to produce finite-state models that are amenable to model-checking verification tools [Dwyer et al., 2001]. (9) Switching from requirements-based design to model-based design helps produce state descriptions [Busser et al., 2002]. (10) Build an observer to estimate the system states. This technique is common in the field of Linear Systems Theory [Szidarovszky and Bahill, 1998]. It is analogous to the following technique. (11) Add extra outputs (test points) so that the states can be identified by examining the outputs.

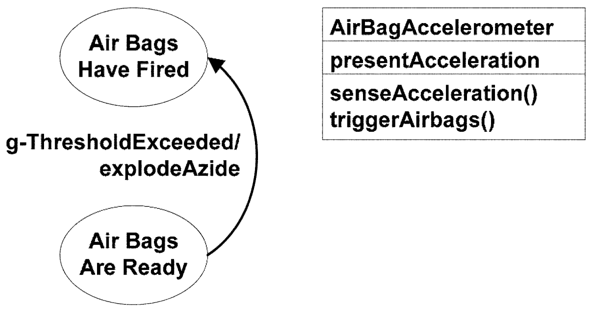


Figure 1. State machine diagram (left) and a class diagram (right) showing how states are represented. The class diagram has three compartments containing the class name (top), attributes in this case indicating the state (middle) and functions (bottom). Copyright © 2004, Bahill, from http://www/sie.arizona.edu/sysengr/slides/, used with permission.

Example of Adding Outputs. In the old days, accelerometers from automobile air bag systems indicated their state as Ready or Fired. At the same time, missile manufactures spent thousands of dollars on g-switches that indicated when a missile had accelerated away from its platform. Then someone added an extra output to the accelerometers, an output that indicated the state of the accelerometer—its present acceleration. Missile manufactures then started using these inexpensive accelerometers in missile safing systems instead of expensive custom made g-switches. Figure 1 shows how the state of an object is represented in a UML state machine diagram and with the values of its attributes in a class diagram.

3. SOMETIMES IT WORKS

Input/output (I/O) equivalence cannot be used to prove the equivalence of two dynamic systems. But many engineers say that this is how they do it. Whenever a lot of people say that they do something that cannot be done, it is useful to examine exactly what they are doing. First, they are using input/output behavior to prove system equivalence and *sometimes* it works. So, next we want to show under what circumstances it works.

In the field of Digital Design (e.g., computer design), the two basic types of systems are called combinational and sequential [Hill and Peterson, 1981; Katz, 1994]. In the field of cybernetics, von Foerster [1982] called them trivial and nontrivial systems. In this paper, we

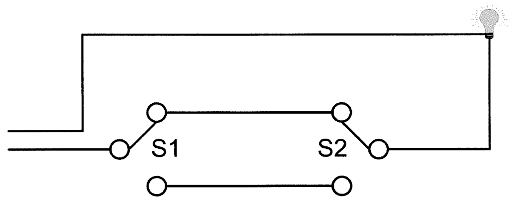


Figure 2. Wiring diagram for a three-way light system. Copyright © 2004, Bahill, from http://www/sie.arizona.edu/sysengr/slides/, used with permission.

will call them, respectively, memoryless and dynamic. In memoryless systems, the output depends only on the present inputs, whereas, in dynamic systems, the output depends on the sequence of previous inputs. Memoryless problems can be modeled and implemented as dynamic systems. But not vice versa.

Consider the household three-way light system of Figure 2 with one light and two switches, one at each end of a hallway. Define the inputs to be the position of the switches {up, down}. We can create the dynamic model of Figure 3 for this three-way light system. For simplicity, input combinations that do not produce a change of state are not shown; also, simultaneous changes of both switches are not shown. Given the present state and an input trajectory (sequence of inputs), we can compute the state trajectory. Therefore, the three-way light system can be modeled as a dynamic system as in Figure 3 and it can be implemented with flip-flops.

However, in actuality, the three-way light system is a memoryless *problem*: It does not *need* a dynamic *solution*. The present state depends only on the present positions of the switches. The system can be modeled with the truth table of Table I. From Table I, we can derive the following Boolean equation:

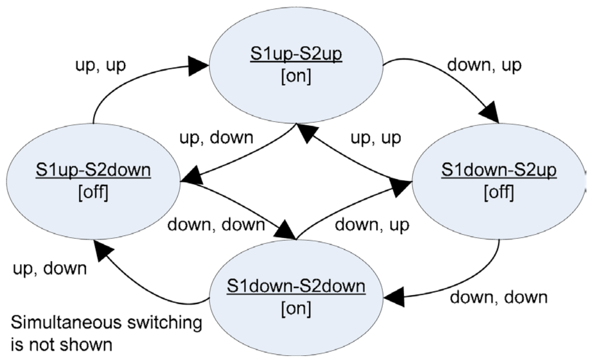


Figure 3. State machine model for a three-way light system. Copyright © 2004, Bahill, from http://www/sie.arizona.edu/sysengr/slides/, used with permission.

Table I. A Truth Table

S2	S1	Light
down	down	on
down	up	off
up	down	off
up	up	on

$$Light = \overline{S1} \cdot \overline{S2} + S1 \cdot S2.$$

This equation (or model) can be implemented using only AND and OR gates. This memoryless system implementation is simpler than the dynamic system implementation presented in Figure 3. Furthermore, the equivalence of this memoryless model with the physical system can be proven using *only* input/output behavior.

Contrast this three-way light system with a three-way lamp that can be off, on at 50 W, on at 100 W or on at 150 W. Turn the switch 90°, and the lamp is on dimly. Turn it again, and the lamp is on with medium brightness. Turn it a third time and the lamp is on brightly. A final turn, turns the lamp off. The behavior of this system (with its only input of turn the knob 90°) clearly depends on its previous state: It is a dynamic problem and it requires a dynamic solution. The equivalence of the dynamic solution (model) with the physical system cannot be proven using only input/output behavior.

Therefore, in the real world, there are memoryless problems that should have memoryless solutions: However, they could also have dynamic solutions. An engineer *can* prove the equivalence of these memoryless problems using only input/output behavior. However, real-world dynamic problems must have dynamic solutions. And the equivalence of these solutions *cannot* be proven using only input/output behavior.

In this paper, we have shown that an engineer cannot prove equivalence of two *dynamic* systems using only input/output behavior. But there are many engineers who say that they have proven the equivalence two systems using only input/output behavior. We suggest that their systems were merely memoryless problems. And for *memoryless* problems, equivalence can be proven using only input/output behavior.

4. WHAT TYPE OF SYSTEMS HAVE STATES AND WHAT TYPE OF MODELS NEED STATES?

In order to use COTS products, we must have a valid model of the system, and that model must have observable states in order to prove that the model conforms to the system design.

Before analyzing models of systems, we need a few definitions. A *system* is a combination of interacting elements that performs a function not possible with any of the individual elements. The elements can include hardware, software, bioware, facilities, policies, and processes. A system accepts inputs, over which it has no direct control, and transforms them into outputs. A system should have a well-defined boundary. Fitting this single definition are many types of systems, some have states and some do not.

Systems can be categorized as memoryless or dynamic. In a *memoryless system*, the outputs depend only on the present values of its inputs, whereas, in a *dynamic system*, the outputs depend on the present and past values of its inputs. For dynamic systems, we must define the concept of a state.

The *state of a system* makes the system's history irrelevant. The state of the system contains all of the information needed to calculate responses to present and future inputs without reference to the past history of inputs and outputs. The state of the system, the present inputs, and the sequence of future inputs allow computation of all future states (and outputs). For example, the present balance of your checking account can be the state of that system. This state could have gotten to its present value in many ways, but when you write a check, that history is irrelevant. But, of course, your model depends on how you define your system. Credit agencies would want to know how many times you had a negative balance and the bank would want to know your daily balance in order to calculate your interest.

Some dynamic systems are modeled best with *state equations* while others are modeled best with *state machines*. State-equation systems are modeled with 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. 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 computer systems are modeled with finite state machines.

At each instant of time, a dynamic system is in a specific state. State-equation systems can have one or many state variables: At any time, the system's state is defined as the unique values for each of the state variables. State-machine systems can be modeled with one or many concurrent state machines: at any time, each of the concurrent state machines must be in one and only one state. A *state* is a unique snapshot that (1) is specified by values for a set of variables, (2) characterizes the system for a period of time, and (3) is different from other states. Each state is different from other states in either the inputs it responds to, the outputs it produces or the transitions it takes. A transition is a response to an input that may cause a change of state.

Some systems require both state machine diagrams and state equations. The UML models state-machine systems with state machine diagrams. It does not have a specific diagram for modeling state-equation systems. Instead, the state equations are put into the use case package [Gomaa, 2000: 598]. In SysML [2006] state equations are modeled in parametric diagrams. State machines can be organized in hierarchies made up of states, superstates, substates, concurrent states, and history states [Harel and Naamad, 1996]. A system could be composed of several superstates each of which is made up of several states. In this context, a mode would be a superstate, although a reviewer has said that MIL STD 490 treated modes in the opposite fashion saying that states were composed of modes.

In modeling a system, one of the most important tasks is deciding whether the system is memoryless or dynamic. First, let us consider some state-equation systems. An ideal spring is a memoryless system, because the output position depends only on the input force applied, whereas an ideal mass is a dynamic system, because the output position depends on the applied input force as well as the initial position and velocity. An ideal resistor is a memoryless system, because the output voltage depends only on the input current, whereas a resistor-capacitor system is dynamic, because the output voltage depends on the input current as well as the initial capacitor voltage.

Let us now consider some well-defined game systems and decide whether they are memoryless or dynamic. Roulette is a memoryless system. You spin the wheel, look at the input, and pay the output. State lotteries are also memoryless systems. The State does not care who won last week: It just draws the numbers and pays the winners. Flipping a coin is memoryless. The card game Show Low is a memoryless game. Each player cuts the deck and shows his card. The player showing the lowest card wins. The city of Show Low in Arizona was named after a famous Show Low game. A padlock is a memoryless system, whereas a combination lock is a dynamic system. The game of Craps requires knowledge of "the point." So it is a dynamic system, but because it requires memory of only one number, it is one of the simplest dynamic games.

Often a question will help you discover the state information that a dynamic system must store in its memory. For instance, if you are playing a game of chess by mail, "What state information does your opponent need in order to respond to your move?" He or she would need to know the position of every piece on the board, whose turn it is, whether or not each person has castled, and about passed pawns. We will now estimate the information complexity of this game by suggesting how much memory this would require. The location of each piece could be prescribed by noting its row and column. For example white's bishop in column d, row 5 would be WBd5. There are 32 pieces each requiring 4 bytes. So the game would require about 140 bytes. Near the end of a game, individual moves would also have to be recorded to possibly indicate a stalemate.

For checkers you would need to record the color and type of each piece on the 32 numbered dark squares. Information complexity is 96 bytes.

If a baseball game were called due to rain, "What state information would you have to store in order to restart it at a later date?" You would need to store total runs scored by the visiting team, total runs scored by the home team, the inning, whether it is the top or the bottom of the inning, last batter for the visiting team, last batter for the home team, outs in this half of the present inning, balls on present batter, strikes on present batter, name of runner on first base, name of runner on second base, name of runner on third base, and a list of players who have been removed from the game (this is important, because in baseball a removed player cannot return). You would need all this state information in order to restart a rain-delayed game. For information complexity, this is about 120 bytes. If you wish to go into excruciating detail, you could also consider the batting order and the number of times each manager and coach has gone out to talk to the pitcher in the inning. For Little League games you could list how many innings each pitcher pitched (pitchers are limited to six innings per week) and the children who were in attendance (children who were not there for the original game cannot play in the resumed game).

Poker is a very complex dynamic system. Before you make a bet, you should know all of the cards that have been played, all of the bets that have been made, and the state of each opponent's mind. To assess their states of mind, you must read their body language and vocal patterns and use this in combination with their previous betting behavior. Poker requires astute observation and a large memory to store a large number of states.

In contrast, Blackjack is a much simpler dynamic system, because you are only playing against the dealer and the dealer must abide by strict rules. So the only state information you need is the cards that have been played. For information complexity, on average 26 cards would have been played and you can record each card with 2 bytes of information. So information complexity is about 50 bytes. Table II summarizes the state behavior of these games.

Of course, all of the above statements depend on how you define your system's boundaries. If you want to model *the amount of money* in the state lottery pot, then you must know if someone won last week. In the game of roulette, if you can measure the state (speed and position) of the wheel and the state (speed and position) of the ball while they are spinning, then you can use physics to predict where the ball will land. In the late 1970s students at the University of California at Santa Cruz successfully built and used such a computer system and won at Las Vegas.

A computer is obviously a dynamic system. If you buy a new computer or install a new operating system, you should record the parameters of your dialup modem, favorite URLs of your browser, custom dictionary of your spelling checker, version, and preferences for all software. These partially capture the state of your previous system. The state of a computer is sometimes called the system configuration.

Occasionally, your personal computer may put up a window saying, "The application has committed an error. Please tell Microsoft about this problem." What are they sending to Microsoft? State information, because Microsoft knows that they can only fix their programs if they have state information.

Table II. For Which Systems Are States Needed?

Game	Are States Needed?	Needed State Information
Roulette	No	None
Coin Toss	No	None
Lottery	No	None
Craps	Yes	The point
Blackjack	Yes	The cards that have been played from the deck
Checkers	Yes	Identity of piece on each square, Who's turn
Baseball	Yes	Batter, inning, count, score, etc.
Chess	Yes	Position of each piece, Castling, Stalemates, Passed pawns, Who's move
Poker	Yes	Bets, Cards in deck, Previous behavior

In software systems, services should not have state behavior [Evans, 2004: 104–108]. Some objects will have states and when they are created: they must be deliberately put into their desired initial states [Evans, 2004: 139].

Almost any system containing a microprocessor will be a dynamic system: microwave ovens, cars, watches, calculators, printers, telephones. Simple models for most inanimate objects will be memoryless systems: chairs, desks, mugs, staplers, bats, and balls. But, of course, your purpose in modeling the system may force you to sometimes use a dynamic model. For example, it might be important to know if the stapler had staples in it. The coefficient of restitution of a bat-ball collision depends upon the internal temperature of the ball, and this would depend on its history, for example, if it had recently been stored in a freezer.

We now want to consider some commercial database systems. Are DOORS and Excel essentially equivalent systems? Yes, for most practical purposes. They have different features, but for creating a simple requirements system, they are equivalent, because the programs themselves do not have significant state behavior. How would you prove the equivalence of a DOORS requirements system and an Excel requirements system? First, you would compare the contents of each memory cell. If they were all equal, then the databases would be equivalent. Then you could merge two Excel databases and two DOORS databases and recheck for equivalence. Then you could delete of all requirements containing a certain keyword from an Excel database and a DOORS database and then compare the resulting databases of each system requirement by requirement. Similar statements should hold for SQL, Sybase, and Oracle requirements systems.

The above paragraph about DOORS and Excel is different from the rest of this paper. The rest of this paper was precise. This paragraph is mushy. It was put in to show the type of decisions engineers frequently make. Most engineers would be willing to upgrade an Excel spreadsheet to a DOORS database, and they would seldom get into trouble doing so, although they would not have proven that such an upgrade would be valid.

DOORS, as its name (Dynamic Object-Oriented Requirements System) implies, is a dynamic system. It has state information. So going from an Excel requirements system to a DOORS system would be all right. But in going backwards from a DOORS system to an Excel requirements system, you would probably lose the state information.

The state of a system is a modeling concept. We can never accurately know the complete, exact state of a physical system. The Heisenberg Uncertainly Principle says, we cannot simultaneously measure the exact position and velocity of an object. Einstein said, "So far as the theorems of mathematics are about reality, they are not certain: so far are they are certain, they are not about reality." A model is a simplified representation of some aspect of a real system. Most systems are impossible to study in their entirety, but they are made up of hierarchies of smaller subsystems that can be studied. Herb Simon [1962] discussed the necessity for such hierarchies in complex systems. He showed that most complex systems are decomposable, enabling subsystems to be studied outside the entire hierarchy. For example, when modeling the movement of a pitched baseball, it is sufficient to apply Newtonian mechanics considering only gravity, air resistance, velocity of the ball, and spin of the ball. One need not be concerned about electron orbits or the motions of the sun and the moon. Forces that are important when studying objects of one order of magnitude seldom have an effect on objects of another order of magnitude. As a consequence, we never know the complete exact state of any physical system. Therefore, our comments about the state of a system are actually comments about some model of that system. Our comment about the need for observable states means, "Do the outputs of the physical system and the model provide an unambiguous description of the state of the model?"

5. FAMOUS FAILURES

The systems presented in Table II are well known and have well-defined rules. Next, we investigated some less well-defined systems [Moody, Chapman, Van Voorhees and Bahill, 1997]. In particular, we looked at the 23 famous failures studied by Bahill and Henderson [2005] and asked, "Could the failure have been caused by mistakes in modeling the states or modes of the system?" The following is a description of some of these systems and Table III provides a summary of the answers to this question.

RMS Titanic had poor quality control in the manufacture of the wrought iron rivets. In the cold water of April 14, 1912, when the Titanic hit the iceberg, many rivets failed and whole sheets of the hull became unattached. They did not build the ship right; therefore, verification was bad. An insufficient number of lifeboats was a requirements development failure. However, the Titanic satisfied the needs of the ship owners and passengers (until it sank), so validation was all right [*Titanic,* 1997]. There is no evidence that the concept of states existed a century ago.

The **Tacoma Narrows Bridge** was a modification of an older design. But the strait where they built it had

Table III. Some Famous Failures

System Name	Year	Could the failure have been caused by mistakes in modeling the states or modes of the system?
RMS Titanic	1912	No
Tacoma Narrows Bridge	1940	Yes
Apollo-13	1970	No
GE refrigerator	1986	Yes
Chernobyl Nuclear Power Plant	1986	Yes
Hubble Space Telescope	1990	No
Ariane 5 missile	1996	Yes
Lewis Spacecraft	1997	Yes
Mars Climate Orbiter	1999	Yes
Mars Polar Lander	2000	No

strong winds: the bridge became unstable in these crosswinds and it collapsed. The film of its collapse is available on the Web: It is well worth watching [Tacoma-1, 2006; Tacoma-2, 2006]. The design engineers reused the requirements for an existing bridge: so these requirements were up to the standards of the day. The bridge was built well, so verification was all right. But it was the wrong bridge for that environment, a validation error [Billah and Scanlan, 1991]. A well-constructed bridge appears to be a static structure. But it actually has a lot of dynamic behavior. That is why army units walk, not march, across bridges. Galloping Gertie, as it was called, exhibited tremendous dynamic behavior, twisting, turning and undulating in the hour before it collapsed. With 21st century computers, it would be easy to model the dynamic state behavior of such a bridge and therefore predict its collapse.

John F. Kennedy, in a commencement address at Duke University in 1961, stated the top-level goals for the Apollo Program: (1) Put a man on the moon (2) and return him safely (3) by the end of the decade. On **Apollo 13**, for the thermostatic switches for the heaters of the oxygen tanks, they changed the operating voltage from 28 to 65 V, but they did not change the voltage specification or test the switches. This was a configuration management failure that should have been detected by verification [Apollo 13, 1995]. The Apollo 13 did have states and modes, but there is no evidence that they had anything to do with the failure.

General Electric Co. (GE) engineers said they could reduce the part count for their new **refrigerator** by one-third by replacing the reciprocating compressor with a rotary compressor. Furthermore, they said they could make it easier to machine, and thereby cut manufacturing costs, if they used powdered-metal instead of steel and cast iron for two parts. However, powdered-metal parts had failed in their air conditioners a decade earlier. [Chapman, Bahill, and Wymore, 1992: 19]

Six hundred compressors were "life tested" by running them continuously for 2 months under temperatures and pressures that were supposed to simulate 5 years' actual use. Not a single compressor failed, which was the good news that was passed up the management ladder. However, the technicians testing the compressors noticed that many of the motor windings were discolored from heat, bearing surfaces appeared worn, and the sealed lubricating oil seemed to be breaking down. This bad news was not passed up the management ladder!

By the end of 1986, GE had produced over 1 million of the new compressors. In July of 1987 the first refrigerator failed; quickly thereafter came an avalanche of failures. The engineers could not fix the problem. In the summer of 1988 the engineers reported that the two powdered-metal parts were wearing excessively, increasing friction, burning up the oil and causing the compressors to fail. GE management decided to redesign the compressor without the powdered-metal parts. In 1989, they voluntarily replaced over one million defective compressors.

The biggest mistakes causing this failure were specifying the powdered-metal parts and bad modeling of the accelerated life-cycle testing. In other words, the model of the state behavior of the motors poorly matched the actual state of the physical system.

The **Chernobyl Nuclear Power Plant** was built according to its design, but it was a bad design: Validation was wrong. They had poor configuration management evidenced by undocumented cross-outs in the operating manual. Human error contributed to the explosion. Cover up and denial for the first 36 hours contributed to the disaster. This is our greatest failure: It killed hundreds of thousands, perhaps millions, of people. Here are references for the U.S. Nuclear Regulatory Commission summary [Chernobyl-1, 2006], a general Web site with lots of other links [Chernobyl-2, 2006], a BBC report [Chernobyl-3, 2006], and for some photos of what Chernobyl looks like today [Chernobyl-4, 2006]. Just before the explosion, the human operators thought that the reactor was in one state when it was actually in another. The mental models of the nuclear reactor operators were wrong.

The **Hubble Space Telescope** was built at a cost of around 1 billion dollars. The requirements were right. Looking at the marvelous images we have been getting, in retrospect we can say that this was the right system. But during its development the guidance, navigation, and control (GNC) system, which was on the cutting edge of technology, was running out of money. So they transferred money from Systems Engineering to GNC.

As a result, they never did a total system test. When the Space Shuttle Challenger blew up, the launch of the Hubble was delayed for a few years, at a cost of around 1 billion dollars. In that time, no one ever looked through that telescope. It was never tested. They said that the individual components all worked, so surely the total system would work. After they launched it, they found that the telescope was myopic. Astronauts from a space shuttle had to install spectacles on it, at a cost of around 1 billion dollars. [Chapman, Bahill, and Wymore, 1992: 16]. Modeling of the states seems to have had no part in the failure.

The French Ariane 4 missile was successful in launching satellites. However, the French thought that they could make more money if they made this missile larger. So they built the **Ariane 5**. It blew up on its first launch, destroying a billion dollars worth of satellites. The mistakes on the Ariane 5 missile were (1) reuse of software on a scaled-up design, (2) failure to test this software in a new mode that would become manifest due to the larger engines, (3) allowing the accelerometer to run for 40 s after launch, (4) not flagging as an error the overflow of the 32-bit horizontal-velocity storage register, and (5) allowing a CPU to shutdown if the other CPU was already shutdown. The requirements for the Ariane 5 were similar to those of the Ariane 4: So it was easy to get the requirements right. They needed a missile with a larger payload, and that is what they got: So that part of validation was all right. However, one danger of scaling up an old design for a bigger system is that it might produce a bad design, which is a validation mistake. Their failure to test the scaled-up software in its new states and modes was a verification mistake [Kunzig, 1997].

The **Lewis Spacecraft** was an Earth-orbiting satellite that was supposed to measure changes in the Earth's land surfaces. But due to a faulty design, it only lasted 3 days in orbit. "The loss of the Lewis Spacecraft was the direct result of an implementation of a technically flawed Safe Mode in the Attitude Control System. This error was made fatal to the spacecraft by the reliance on that unproven Safe Mode by the on orbit operations team and by the failure to adequately monitor spacecraft health and safety during the critical initial mission phase" [Lewis Spacecraft, 1998]. The failure investigation board wrote that the failure was caused by a faulty mode.

On the **Mars Climate Orbiter**, the prime contractor, Lockheed Martin, used English units for the satellite thrusters while the operator, JPL, used SI units for the model of the thrusters. Therefore, there was a small mismatch between the space-based satellite and the ground-based model due to different round-off effects. Because the solar arrays were asymmetric, the thrusters had to fire often, thereby accumulating error between the state of the satellite and the state of the model. This caused the calculated orbit altitude at Mars to be wrong. Therefore, instead of orbiting, it entered the atmosphere and burned up [Mars Program, 2000; NASA, 2000]. There was a mismatch between the space-based satellite and the ground-based model: This is a validation error. This failure is obviously due to mistakes in modeling the system state.

On the **Mars Polar Lander**, "spurious signals were generated when the lander legs were deployed during descent. The spurious signals gave a false indication that the lander had landed, resulting in a premature shutdown of the lander engines and the destruction of the lander when it crashed into the Mars surface … It is not uncommon for sensors … to produce spurious signals… During the test of the lander system, the sensors were incorrectly wired due to a design error. As a result, the spurious signals were not identified by the system test, and the system test was not repeated with properly wired touchdown sensors. While the most probable direct cause of the failure is premature engine shutdown, it is important to note that the underlying cause is inadequate software design and systems test" [Mars Program, 2000] [Blackburn, Busser and Nauman, 2002]. Modeling of the system states and modes seems to have had no part in the failure.

The emphasis in most of this paper was that observable states are necessary in order to prove system equivalence. Another reason for providing observable states is to help a team to better understand the behavior of a complex system. Safe operation of a complex system is based on the ability to explain system behavior. Explanations of system behavior are based on functions and states [Rouse, Cannon-Bowers, and Salas, 1992].

6. DISCUSSION

In the first half of the 20th century, the reigning model for **human decision-making** was the theory of subjective expected utility. Utility is a measure of the happiness or satisfaction a person receives from a good or service. Utilities are numbers that express relative preferences using a particular set of assumptions and methods. Subjective expected utility combines two subjective concepts: utility and probability. The subjective expected utility is the product of the utility and the probability of that event occurring. Subjective expected utility theory models human decision-making as maximizing subjective expected utility (1) maximizing, because people choose the set of alternatives with the highest total utility, (2) subjective, because the choice depends on the decision maker's values and preferences, not on reality (e.g. advertising improves subjective perceptions of a product without improving the product), and (3) expected, because the expected value is used. A person who maximizes subjective expected utility is said to be rational. This is a first-order model for human decision-making.

The term *satisficing* was coined by Noble Laureate Herb Simon [1955]. When making decisions there is always uncertainty and insufficient time and resources to explore the whole problem space. Therefore, people cannot make rational decisions. Simon proposed that people do not attempt to maximize the expected utility of the alternatives. Instead, they search for alternatives that are good enough, alternatives that satisfice.

Our situation with observable states seems to be analogous to the situation Simon was in during the 1950s. We know that proving equivalence of systems can only be done with observable states. But for complex COTS systems we know that the states are too numerous to elaborate and vendors are not going to give us access to the states. Therefore, we need a satisficing alternative. For systems that are not safety critical, perhaps we could identify a crucial subset of the system states and then prove equivalence only of that partition. This subset of states might be identifiable using information theory, interactive abstraction [Dwyer et al., 2001; Liu, 2000] or criteria hierarchies [Daniels, Werner and Bahill, 2001].

In the 1940s and 1950s control systems engineers developed **transfer function analysis** to design and model systems. It was often said that the impulse response (or the step-response, etc.) would completely characterize the system. However, this proved to be untrue, and modern control theory with its concept of system state became the preferred tool for analyzing and modeling systems [Szidarovszky and Bahill, 1997]. The following two transfer functions have identical input/output behavior, but their dynamics are quite different:

$$TF_1 = \frac{Output(s)}{Input(s)} = \frac{1}{s + p_1},$$

$$TF_2 = \frac{Output(s)}{Input(s)} = \frac{s + z_1}{(s + p_1)(s + p_2)}, \text{ where } p_2 = z_1 > 0.$$

For physical systems modeled with such transfer functions, system-2 has a pole in the right-half of the *s*-plane! The system is unstable. Transfer functions and impulse responses are often inadequate models for systems: States must be used.

7. SUMMARY

If we are to use a COTS product in a new system design, then we will have to find a way to prove that the COTS product is equivalent to its design. First, we need to decide if the COTS product is memoryless or dynamic. If it is memoryless, we can apply a string of inputs to both the COTS product and its design model and observe the string of outputs. If they are the same, then the COTS product is equivalent to the design model. However, if the COTS product is a dynamic system, then we must ask if the states are observable. If they are, we can design system experiments where we start the COTS product and the design model in equivalent states, apply a string of inputs, and observe the resulting string of states. If we do this rigorously, we can prove that they are equivalent. However, if the states are not observable, then we need some other technique to assure that the behavior of the COTS product is acceptable.

In industry, many dynamic systems are tested using only test inputs and measuring outputs without regard to system states. Evidentially this is relatively successful, if the input space is suitably sampled and many checks and balances are used. But nagging doubts have caused many engineers to look for other confirmatory techniques for system testing.

Here are some techniques that have been used in lieu of proving system equivalence. (1) Create multiple reset (or test) states and prove I/O equivalence with respect to an initial state pair for all of them. (2) Implement built-in self-tests. (3) Use regression testing. (4) Define pre and post conditions. (5) Only use COTS products in places where you do not care about observable states. (6) Put a wrapper around COTS products. (7) Record the mode the system was in when the event of interest occurred. (8) Provide abstracting software to produce finite-state models that can be analyzed by model-checking verification tools. (9) Build an observer to estimate the system states. (10) Add extra outputs so that the states can be identified by examining the outputs.

The equivalence of two systems can be proven using input/output behavior for memoryless systems, but not for dynamic systems. The equivalence of dynamic systems can only be proven by observing the string of states, which means that the systems must have observable states. So before trying to prove system equivalence, we must decide whether the systems are memoryless or dynamic; and if they are dynamic, we must ensure that the states are observable.

All systems should have the capability of displaying critical system states. This capability can be decomposed into system, hardware, and software requirements. System, hardware, and software engineers should be required to design observable states. The required set of observable states would be stated in the derived requirements.

REFERENCES

Apollo 13, produced by Imagine Entertainment and Universal Pictures, Hollywood, CA, 1995.

A.T. Bahill, Use case template, http://www.sie.arizona.edu/sysengr/slides/template.doc, 2006.

A.T. Bahill and S.J. Henderson, Requirements development, verification and validation exhibited in famous failures, Syst Eng 8(1) (2005), 1–14.

Y. Billah and B. Scanlan, Resonance, Tacoma Narrows bridge failure, and undergraduate physics textbooks, Am J Phys 59(2) (February 1991), 118–124, see also http://www.ketchum.org/wind.html.

M.R. Blackburn, R. Busser, and A. Nauman, Mars Polar Lander fault identification using model-based testing, Proc Eighth Int Conf Eng Complex Comput Syst, 2002.

R.D. Busser, M.R. Blackburn, A.M. Nauman, and T.R. Morgan, Reducing cost of high integrity systems through model-based testing, Digital Avionic Syst Conf, October 24–28, 2004, pp. 6.B.1-1–6.B.1-13.

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

Chernobyl-1, http://www.nrc.gov/reading-rm/doc-collections/fact-sheets/fschernobyl.html, U.S. Nuclear Regulatory Commission, Washington, DC, 2006.

Chernobyl-2, http://www.infoukes.com/history/chornobyl/zuzak/page-07.html, U.S. Nuclear Regulatory Commission, Washington, DC, 2006.

Chernobyl-3, http://www.chernobyl.co.uk/, U.S. Nuclear Regulatory Commission, Washington, DC, 2006.

Chernobyl-4, http://www.angelfire.com/extreme4/kiddofspeed/chapter27.html, U.S. Nuclear Regulatory Commission, Washington, DC, 2006.

J. Daniels, P.W. Werner, and A.T. Bahill, Quantitative methods for tradeoff analyses, Syst Eng 4(3) (2001), 199–212.

Department of Defense (DOD), Directive Number 5000.1, The Defense Acquisition System, Washington, DC, May 12, 2003.

M.B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C.S. Pasareanu, H. Zheng, and W. Visser, Tool-supported program abstraction for finite-state verification, Int Conf Software Eng, 2001, pp. 177–187.

E. Evans, Domain-driven design: Tackling complexity in the heart of software, Addison-Wesley; Boston, 2004.

H. Gomaa, Designing concurrent, distributed, and real-time applications with UML, Addison-Wesley, Reading, MA, 2000.

D. Harel and A. Naamad, The STATEMATE semantics of statecharts, ACM Trans Software Eng Methodol 5(4) (1996), 293–333.

F.J. Hill and G.R. Peterson, Introduction to switching theory and logical design, Wiley, New York, 1981.

R.H. Katz, Contemporary logic design, Benjamin/Cummings, Redwood City, CA, 1994.

R. Kunzig, Europe's dream, Discover 18 (May 1997), 96–103.

Lewis Spacecraft Mission Failure Investigation Board, Final Report, NASA, Washington D.C., 12, February 1998.

W. Liu, Interaction abstraction for compositional finite state systems, SPIN model checking and software verification, Proc 7th Int SPIN Workshop, Stanford, CA, August 30–September 1, 2000, pp. 148–162, http://netlib.bell-labs.com/netlib/spin/ws00/18850150.pdf.

Mars Program Independent Assessment Team Summary Report, NASA, Washington D.C., March 14, 2000.

J.A. Moody, W.L. Chapman, F.D. Van Voorhees, and A.T. Bahill, Metrics and case studies for evaluating engineering designs, Prentice Hall PTR, Upper Saddle River, NJ, 1997.

NASA Faster Better Cheaper Task Final Report, NASA, Washington D.C., 2, March 2000.

NSF Center for Empirically Based Software Engineering, National Science Foundation, Washington, DC, 2005, http://www.cebase.org/www/frames.html.

W.B. Rouse, J.A. Cannon-Bowers, and E. Salas, The role of mental models in team performance in complex systems, IEEE Trans Syst Man Cybernet 22(6) (1992), 1296–1308.

H.A. Simon, A behavioral model of rational choice, Quart J Econom 59 (1955), 99–118.

H.A. Simon, The architecture of complexity, Proc Amer Phil Soc 106 (1962), 467–482.

Software Engineering Institute, http://www.sei.cmu.edu/cbs/index.html, Pittsburgh PA, 2005.

SysML, www.SysML.org, 2006.

Systems and Software Consortium, http://www.systemsandsoftware.org/ssci/default.asp, Herndon VA, 2005.

F. Szidarovszky and A.T. Bahill., "Stability analysis," The electrical engineering handbook, R.C. Dorf (Editor), CRC Press, Boca Raton, FL, 1993, pp. 207–223, 1993, 2nd edition, 1997, pp. 223–238.

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

Tacoma1, http://www.enm.bris.ac.uk/amm/tacoma/tacoma.html, 2006, University of Bristol, UK, .

Tacoma2, http://ketchum.org/bridgecollapse.html, 2006.

Titanic, a Lightstorm Entertainment Production (20th Century Fox and Paramount), Hollywood, CA, 1997.

USC Center for Software Engineering, http://sunset.usc.edu/research/COCOTS/, Los Angeles, CA, 2005.

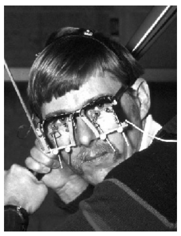
F. von Foerster, Observing systems, Intersystems, Seaside, CA, 1982.

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

A.W. Wymore and A.T. Bahill, When can we safely reuse systems, upgrade systems or use COTS components? Syst Eng 3(2) (2000), 82–95.

Rick Botta is the Director of Systems Engineering for BAE Systems in San Diego. He holds a B.S. degree in Computer Science from California Polytechnic State University, San Luis Obispo. Rick has a quarter century experience in a wide variety of engineering, engineering management and program management roles involving development and integration of complex, software intensive systems. He is a member of INCOSE.

Zach Bahill is a systems engineer with Boeing Integrated Defense Systems in Kent, WA. He earned a B.S. in Electrical Engineering in 2001 from the Department of Electrical and Computer Engineering at the University of Arizona in Tucson.



A. Terry Bahill is a Professor of Systems Engineering at the University of Arizona in Tucson. While on sabbatical from the University of Arizona, he did research with BAE Systems in San Diego. He received his Ph.D. in electrical engineering and computer science from the University of California, Berkeley, in 1975. He holds U.S. Patent No. 5,118,102 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/