Donald G. Firesmith

Advanced Software Technology Specialists (ASTS)
2910 Webster Street, Fort Wayne, IN 46807;
email: 73664.3515@compuserve.com

Object-oriented state modeling using ADM4

THIS ARTICLE SUMMARIZES the state modeling approach of Version 4.5 of the ASTS Development Method (ADM), which includes several major advances over that of ADM3.[1] We begin by defining the basic concepts of state modeling (e.g., states, transitions, triggers) in terms of object-oriented concepts, and then introduce ADM's state transition diagrams and state operation tables, followed by two examples. We then briefly address the issue of the inheritance and documentation of state models, before concluding with a comparison of ADM's state model to those of other methods.

ADM4 is a fourth-generation, general-purpose, object-oriented system development method that emphasizes software development and provides both static architecture and dynamic behavior modeling. While designed for use on large, complex, real-time applications, ADM4 can be easily tailored down for small, simple applications. The primary influences that have shaped ADM's state model include:

- Moore for operations only within states.[2]
- Harel for substates.[3]
- Embley et al. for concurrent states and associated transitions.[4]
- Selic et al. for group transitions involving aggregate states.[5]

STATES

According to ADM, *objects* are models of individual things, either tangible or intangible, in the application domain or its supporting software. To adequately model all relevant aspects of these things, objects have *characteristics* that include both *properties* and *behavior*. Properties include *attributes* that describe the object, *links* capturing relationships to other objects, and *subobjects* as component parts of aggregate objects. Behavior is provided by *operations* and *exception handling*. Objects may also have *invariants*, which are assertions involving the properties that must be maintained by the behavior. Objects communicate via *messages*, each of which typically requests a service provided by one or more operations, and via *exceptions*, which servers use to notify clients of error conditions. Each object is an instance of one or more *classes*, which incrementally provide its definition via inheritance.

Unlike functions, objects and classes have properties, and thus their behavior upon receiving a message need not depend solely on the message and its parameters. Their behavior may also depend on their past history of messages and on the associated operations, which may have modified the values of their properties. An object encapsulating properties may therefore be a *state machine* in the mathematical sense, and collaborating objects therefore may be modeled as communicating state machines.

Because all instances of the same class have the same definition, they all share the same state model. However, different instances have different state machines because the values of their properties may be different. For example, different temperature sensor objects belonging to the same temperature sensors class may occur in different parts of the application (i.e., have links to different objects) and typically measure different temperatures (i.e., have different attribute values). A class with class properties may also have a state machine that differs from that of its instances. Note that the localization of state machines within objects greatly minimizes the combinatorial explosion of states that has historically occurred on large projects driven by a single state machine.

One or more properties may (or may not) determine the overall behavior of its object or class. These properties can determine which messages may be received and sent, which operations may execute, how these operations execute, and which exceptions are raised. Any such property that determines the overall behavior of its object or class is called a *state property*. All combinations of these *state attribute* values, *state links*, or states of *state subobject* that produce the same overall behavior constitute a single *state* of the associated object or class. A state is therefore an equivalence class of state property values that models a condition, mode, or status of its object or class during which specific rules of behavior apply.

While the exact behavior of an object or class depends *quantitatively* on the specific values of its state properties, the object or class behaves *qualitatively* the same for all state properties belonging to the same state and behaves qualitatively different for

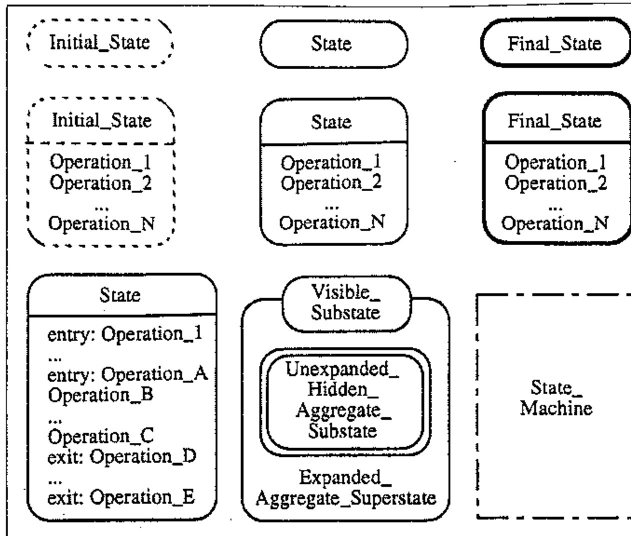


Figure 1. Icons for nodes on states on state transition diagrams.

state properties belonging to different states. Thus, although every combination of attribute values, links, and components is allocated to one or more states, every state need not consist of a single, unique combination of attribute values, links, and components.

Like Harel,[3] ADM recommends the use of aggregate states to simplify state modeling and further minimize the traditional combinatorial explosion of states, already limited by the localization of states within individual objects. An *aggregate state* is a generalized *superstate* that can be decomposed into two or more specialized *substates,* whereas an *atomic state* cannot. For example, the

ADM4 emphasizes software development and provides both static architecture and dynamic behavior modeling.

simple traffic signal can be in either the aggregate state Functioning or the atomic state Malfunctioning, whereby the Functioning state can be decomposed into the atomic states Green, Amber, and Red. The concept of aggregate state is recursive, whereby the substates of aggregate states may themselves be aggregate states. Substates of aggregate states may be visible or hidden. *Visible substates* are substates of an aggregate state into which states outside the aggregate state may directly transition, whereas *hidden substates* may only be transitioned into from another substate of the same aggregate state. As specializations, substates inherit the features (e.g., transitions) of their superstates and may add new features (e.g., transitions) not found in their superstate.*

* This is similar to but different than inheritance between classes, which also affects state modeling. Figure 7 illustrates this in that the two substates (Disabled and Enabled) of the Functioning superstate both inherit the update <Hardware_Failed> transition to the Malfunctioning state. The Enabled substate also introduces the calibrate transition to the Calibrated state.

An *initial state* is the initial atomic state of an object or class upon instantiation. Most objects and classes have only a single initial state, although multiple initial states may exist, in which case the choice of initial state would depend on the parameters of instantiation. A *final state* is an aggregate or atomic state from which an object or class cannot transition.

ADM4 uses the icons in Figure 1 to graphically depict states and state machines. A state is drawn as a rounded rectangle, and unexpanded aggregate states that hide their substates are signified by a double boundary. Visible substates are drawn straddling the boundary of their aggregate state, whereas hidden substates are drawn nested inside their aggregate state. An initial state is drawn with a dashed boundary, whereas a final state has a thick boundary. All states are labeled with a meaningful identifier that is unique within its enclosing state.

An atomic state can optionally be annotated with a list of the operations that *may* execute when the object or class is in that state. It may be impossible to specify the operations that execute in each state (e.g., those that occur upon entering the state and those that occur prior to leaving the state) because (1) the behavior of objects may not be deterministic due to concurrent operations, the random arrival of messages, and the potential for direct updating of attribute values by hardware; and (2) determining which operations are allowed may depend upon multiple simultaneous states. Nevertheless, where practical the operations permitted within a state may be decomposed into three categories and so annotated in the optional list of operations in the state icon. Thus, a state may be labeled with those operations that always:

- begin execution upon transition into the state (such operations are annotated with the word *entry*). These entry operations may be annotated with the specific incoming transitions that trigger them.
- execute while in the state (no annotation required)
- complete execution before transition out of the state (such operations are annotated with the word *exit*). These exit operations may be annotated with the outgoing transitions that they trigger.

TRANSITIONS AND TRIGGERS

A *transition* is a change of state and *fires* (i.e., occurs) when an object or class changes state. An object or class transitions to a new state when the values of its state properties change, producing a new overall set of behaviors. For example, a simple traffic signal normally transitions from green to amber to red and then back to green, repeating the cycle. When concurrency permits multiple transitions to occur, then the transitions can be prioritized so that the highest priority transition occurs first.

A *trigger* is defined as the immediate cause of one or more state transitions. Triggers are therefore said to fire transitions. Because of the way states are defined, a trigger is typically an operation that changes the value of one or more state properties. Any operation that can directly cause a state transition of its object or class is called a *modifier*, whereas any operation that cannot is called a

preserver.† A modifier therefore may (1) sufficiently change the value of one or more state attributes, (2) create or destroy one or more state links, or (3) update the state of one or more component subobjects. A modifier may do this during normal execution (i.e., be a normal trigger) or during the execution of an associated exception handler (i.e., be an exceptional trigger). Because there is often a one-to-one relationship between a message and its associated operation, the message is often misidentified as the trigger when it is actually the associated operation that changes the state of the object or class. A modifier need not change the state of its object or class each time it executes. Sometimes a modifier only modifies the state of its object or class as the result of one or more messages it sends to other objects and classes. These messages may return parameters (e.g., the state of the server object or class) used to determine the new state of the current object or class.

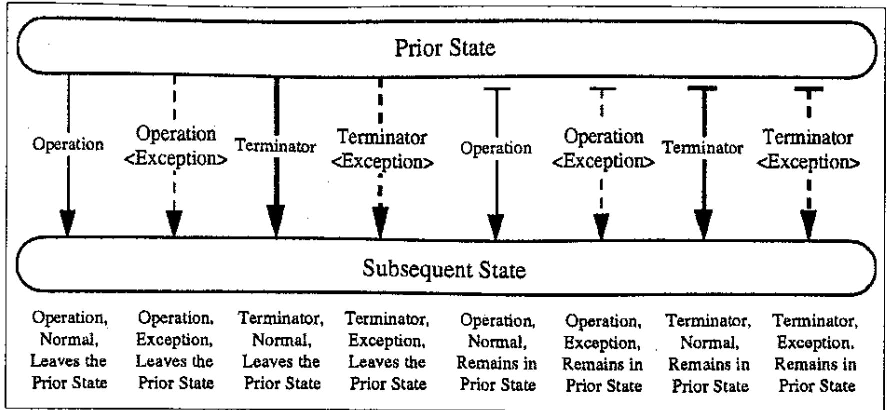


Figure 2. Icons for arcs on transitions on state transition diagrams.

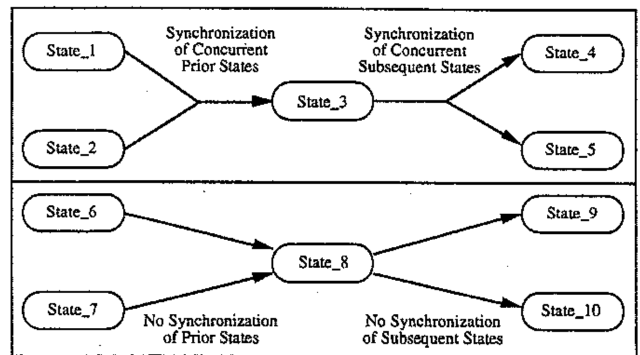


Figure 3. Icons for transitions on state transition diagrams.

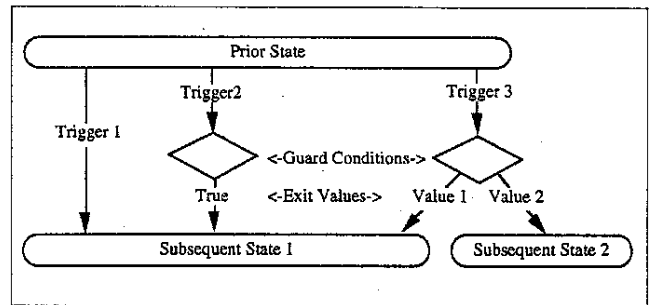


Figure 4. Guard conditions on state transition diagrams.

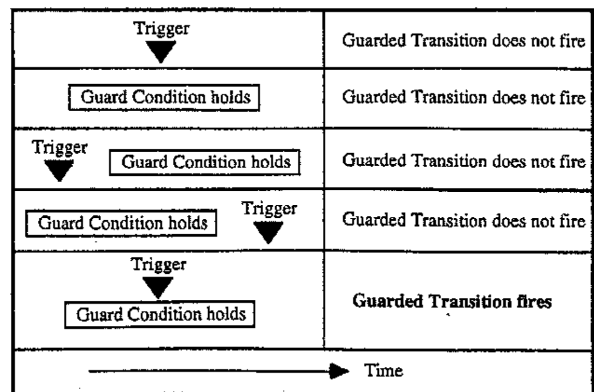


Figure 5. Interaction of triggers and guard conditions.

Because "state" is implemented in terms of the state properties of objects and classes, a state has finite duration and typically occupies a significant interval of time. However, transitions between states are essentially instantaneous relative to the duration of states, and are therefore considered to have only an infinitesimal duration.‡ Thus, an object or class is always considered to be in one or more states at any given time. Because operations also take a finite amount of time to execute, operations are associated with states rather than transitions. This is why ADM4 uses the Moore[2] rather than the Mealy[6] approach to state modeling. This is also why operations are optionally listed in the state icon, but not (except for the trigger operation) on the transition arcs. Allowing objects and classes to perform operations while not in any specific state, as do certain object-oriented methods, is incompatible with the way state is defined in terms of the state properties of objects and classes. Although ADM state transition diagrams (STDs) may look like Mealy STDs because transitions are labeled with trigger operations, the proper interpretation is that the operation starts execution in the prior state, causes the transition by updating one or more state properties, and completes execution in the subsequent state.

In rare cases a state transition also may be caused by a *terminator* of the object or class. A terminator (e.g., hardware device, operating system) may change the value of a state attribute by directly writing to memory. For example, a hardware sensor connected by a serial port may directly update the memory location of the attribute of an associated software sensor. The update of an attribute by an ex-

† This is actually an oversimplification as two other types of operations also exist. A *constructor* creates an instance of a class and initializes its state, whereas a *destructor* destroys an instance of a class.

‡ It usually does not take long to update the value of attributes, create or destroy a link, or update a subobject. However, this tiny duration of a state transition may become important on hard real-time applications. During the transition, outside influences (e.g., messages) are ignored and queued until after the transition, which, if necessary, should occur within a critical region of code.

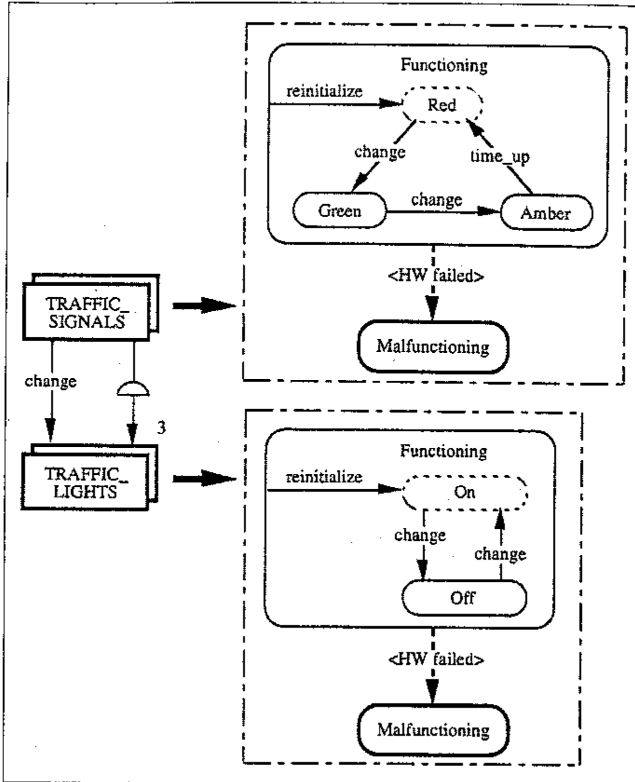


Figure 6. Example state transition diagrams.

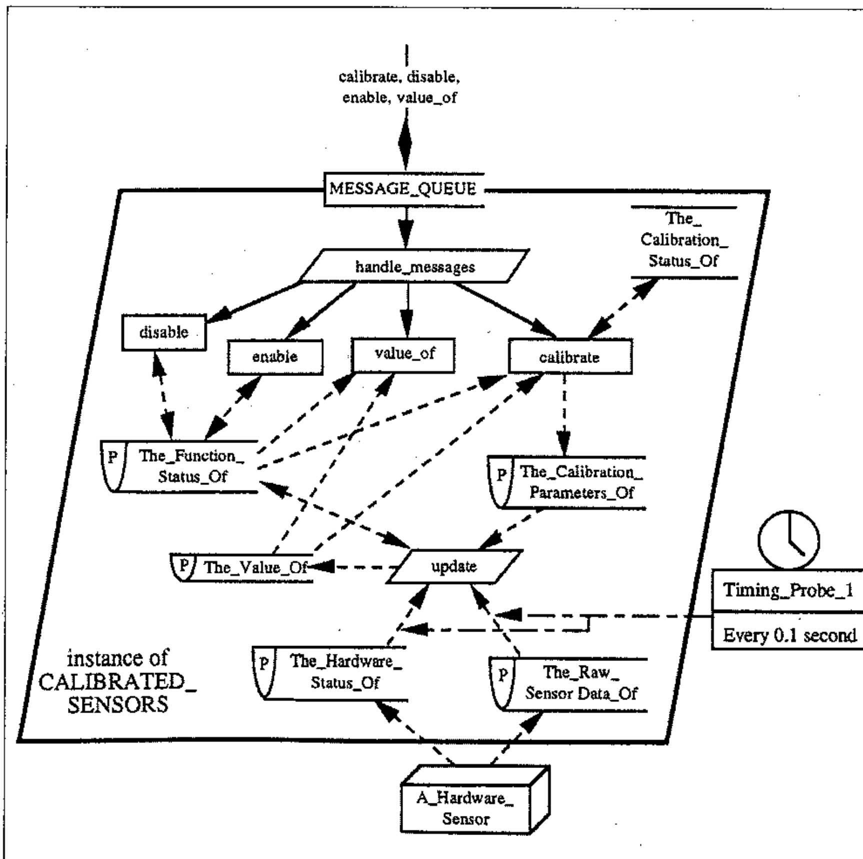


Figure 7. Whitebox interaction diagram for the Calibrated_Sensors class.

ternal entity violates the encapsulation of its object or class and must be well documented, done carefully, and used only where necessary.

A trigger may be either normal or exceptional. A normal trigger is one that occurs during the normal execution of an object or class, whereas an exceptional trigger is one that occurs during the abnormal execution of an object or class (e.g., during exception handling). An exceptional trigger often causes the object or class to transition into an exceptional state.

ADM4 uses the arcs illustrated in Figure 2 to graphically depict state transitions. Transitions are drawn as arrows from the prior state to the subsequent state. Unlike some methods, ADM ignores "transitions" from a state to itself because such "transitions" clutter the diagrams and are better documented elsewhere. The type of arrow used for the transition arc depends on the type of the trigger. If the trigger is the normal execution of an operation or terminator, the transition arc is drawn with a solid line, but if the trigger is the exceptional execution of an operation or terminator, the arc is drawn with a dashed line. If the trigger is an operation, the transition is drawn with a solid line, whereas if the trigger is a terminator, the transition arc is drawn with a thick line. Each transition arc is labeled with the associated trigger(s). Like Embley et al.,[4] ADM allows objects and classes to be in two or more concurrent states, whereby two states are *concurrent* if and only if the object or class can be in both states simultaneously. For example, a person may simultaneously be eating breakfast and reading the newspaper. Also like Embley et al.,[4] ADM allows an object or class to either (1) transition from a prior state, leaving it to enter a subsequent state (left part of Fig. 2) or (2) transition to one or more subsequent states while also remaining in the prior state (right part of Fig. 2).

As illustrated in Figure 3, two concurrent prior states can synchronize and simultaneously transition into a subsequent state (upper left part of Fig. 3), or they may individually transition into the subsequent state (lower left part of Fig. 3). A prior state also can simultaneously transition into multiple synchronized concurrent subsequent states (upper right part of Fig. 3), or a prior state can individually transition into one or the other of the subsequent states (lower right part of Fig. 3).

A transition also may be either spontaneous or non-spontaneous. A *spontaneous transition* occurs on its own without the need for an incoming message and may result from either the execution of a concurrent modifier operation or from the direct updating of a state property of a terminator. A *non-spontaneous transition* requires an incoming message, and most often results from the execution of a sequential modifier operation. Although recognized by ADM4, the difference between spontaneous and non-

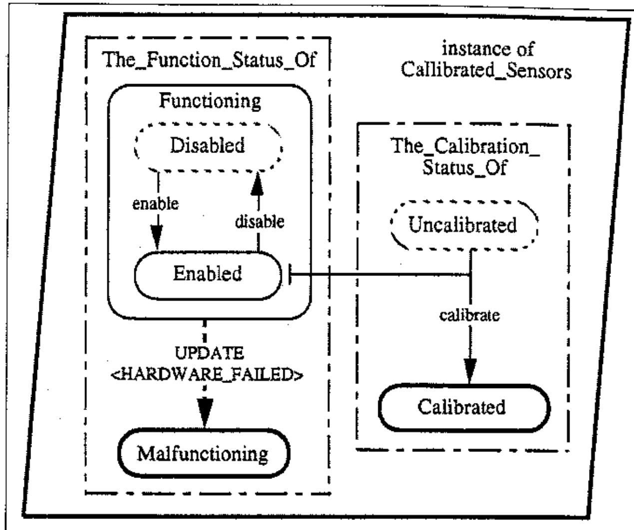


Figure 8. STD for THE_SENSOR Object

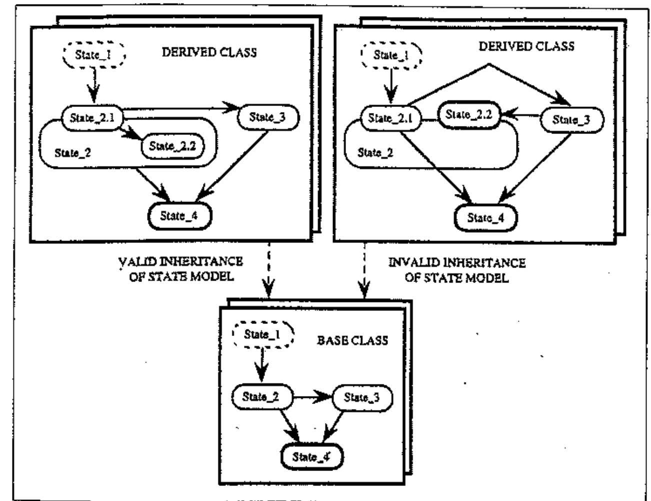


Figure 9. Inheritance of state models.

spontaneous triggers is not explicitly denoted on state transition diagrams.

A *guard condition* (or guard) is an expression involving one or more properties that influence the outcome of one or more state transitions. The guard condition need not involve only state properties. If a trigger of a transition with a guard condition fires, the guard condition is evaluated and the transition occurs if and only if the guard condition evaluates true (for Boolean expressions) or evaluates to the value of the corresponding exit arc of the guard condition (for expressions of enumeration type). As illustrated in Figure 4, a guard is drawn as a diamond with exit values drawn as labeled arcs. Guard conditions are used to fine-tune the control of transitions, and to document cases in which the operation triggering the transition is not the sole cause of the transition (e.g., the parameters returned by a message it sent to another object also influence the transition). A modifier may cause:

- a transition every time it executes (i.e., no guard condition)
- a transition only if the corresponding guard condition evaluates true (i.e., Boolean guard condition)
- transitions to different subsequent states depending on the value of a guard condition of enumeration type.

As illustrated in Figure 5, a transition and its trigger are considered to occur instantaneously. A trigger must occur during the time that the associated guard condition evaluates favorably for the transition to fire, and this typically has a significant duration. If the trigger occurs by itself, or occurs before or after the guard condition evaluates favorably, then the transition does not fire. The transition also does not fire if only the guard condition is satisfied and no trigger occurs.

STDs

If an object or class has a finite number of states, it can be modeled by one or more finite state machines. A state machine, in turn, can be documented using a state transition diagram that illustrates its states and the transitions between them. An STD documents (1) the states of an object or class and (2) the transitions between the states. A states is labeled with its identifier, and a transition is labeled with its trigger. An exceptional transition is also labeled with its exception. A guarded transition is annotated with its guard conditions. Optionally, each state may be labeled with the operations that may execute in that state and each transition may be annotated with the priority of the transition. To simplify the STD, it may be decomposed into separate, possibly interacting sub-STDs (e.g., if multiple state properties or machines exist).

EXAMPLES

As a first example, consider the state models illustrated in Figure 6. A simple traffic signal is an aggregate object consisting of three atomic objects: a green light, an amber light, and a red light. The individual light objects can be in one of three states: on, off, or malfunctioning. The aggregate signal object, however, can be in one of the following four states: green, amber, red, or malfunctioning. Note that although the state of its subobjects influences the state of an aggregate object, its state is not merely the union of the states of its component subobjects (as some methodologists write). New state behavior often occurs at higher levels of abstraction, and the whole is often more than the sum of its parts. Note also that the state of an object determines what messages it can receive, and how the associated operations execute.

Because substates are specializations of their more generalized superstates, they will often respond the same way to the same trigger, producing what has been termed a *group transition.*[5] ADM4 divides group transitions into *group-internal transitions* (e.g., the reinitialize transition) and *group-external transitions* (e.g., the hardware failed transition). A group-internal transition is any transition to an aggregate state as a whole that is forwarded to a specific substate of the aggregate superstate. In this example, it does not matter in which functioning substate the reinitialize operation executes—the machine transitions to the Red substate. Similarly, a group-external transition is any transition from an aggregate state as a whole. In this example, it does not matter in which functioning substate the

Table 1. Example state operation table (SOT) for calibrated sensors.

State Operation Table (SOT) for instances of the class: Calibrated Sensors					
Operation	**Prior States**		**Exception**	**Subsequent States**	
	The_Function_ Status_Of	Calibration_ Status_Of	**Raised by Precondition**	The_Function_ Status_Of	Calibration_ Status_Of
CALIBRATE	Disabled	Calibrated	Disabled	--	--
• Sequential	Disabled	Uncalibrated	Disabled	--	--
• Modifier	Enabled	Calibrated	--	--	--
	Enabled	Uncalibrated	--	--	**Calibrated**
	Malfunctioning	Calibrated	Malfunctioning	--	--
	Malfunctioning	Uncalibrated	Malfunctioning	-	--
HANDLE_	Disabled	Calibrated	--	--	--
MESSAGES	Disabled	Uncalibrated	--	--	--
• Concurrent	Enabled	Calibrated	--	--	--
• Preserver	Enabled	Uncalibrated	--	--	--
	Malfunctioning	Calibrated	Malfunctioning	--	--
	Malfunctioning	Uncalibrated	Malfunctioning	--	--
DISABLE	Disabled	Calibrated	--	--	--
• Sequential	Disabled	Uncalibrated	--	--	--
• Modifier	Enabled	Calibrated	--	**Disabled**	--
	Enabled	Uncalibrated	--	**Disabled**	--
	Malfunctioning	Calibrated	Malfunctioning	--	--
	Malfunctioning	Uncalibrated	Malfunctioning	--	--
ENABLE	Disabled	Calibrated	--	**Enabled**	--
• Sequential	Disabled	Uncalibrated	--	**Enabled**	--
• Modifier	Enabled	Calibrated	--	--	--
	Enabled	Uncalibrated	--	--	--
	Malfunctioning	Calibrated	Malfunctioning	--	--
	Malfunctioning	Uncalibrated	Malfunctioning	--	--
UPDATE	Disabled	Calibrated	--	--	--
• Concurrent	Disabled	Uncalibrated	--	--	--
• Modifier	Enabled	Calibrated	-- Malfunctioning	Enabled **Malfunctioning**	-- --
	Enabled	Uncalibrated	-- Malfunctioning	Enabled **Malfunctioning**	-- --
	Malfunctioning	Calibrated	--	--	--
	Malfunctioning	Uncalibrated	--	--	--
VALUE_OF	Disabled	Calibrated	Disabled	--	--
• Sequential	Disabled	Uncalibrated	Disabled	--	--
• Preserver	Enabled	Calibrated	--	--	--
	Enabled	Uncalibrated	--	--	--
	Malfunctioning	Calibrated	Malfunctioning	--	--
	Malfunctioning	Uncalibrated	Malfunctioning	--	--

hardware failure exception is raised; the machine transitions to the Malfunctioning final state.

As a second example, consider a class of concurrent software sensors that model and periodically read an associated hardware sensor. As illustrated in Figure 7, this class exports[§] four synchronous messages (i.e., calibrate, disable, enable, and value_of) used to calibrate, disable, and enable the sensor and to return the current value of the sensor. These messages are first received by the concurrent handle_messages operation, which ensures mutually exclusive access to the object and, in turn, passes the messages on to the corresponding four sequential operations: calibrate, disable, enable, and value_of. Every 0.1 second (as shown by the timing probe[||]), the concurrent update operation reads the protected[#] hardware status and raw sensor data attributes that are directly updated by the hardware sensor (e.g., via a serial port). The update operation then uses the calibration parameters to convert the raw sensor data into the sensor value.

Calibrated sensors may be either enabled, disabled, or malfunctioning (e.g., if the hardware fails). Calibrated sensors may also be either uncalibrated or calibrated. The state of a calibrated sensor is thus captured by two state attributes: Function_Status and Calibration_Status, which are variables of enumeration type that may take on the associated values. As a whole, calibrated sensors may therefore be in the following 6 (i.e., 3 times 2) combinations of states:

- Disabled + Calibrated
- Disabled + Uncalibrated
- Enabled + Calibrated
- Enabled + Uncalibrated
- Malfunctioning + Calibrated
- Malfunctioning + Uncalibrated

Figure 8 documents the STD for the Calibrated_Sensors class. The state transition diagram is drawn inside the icon for an instance of the Callibrated_Sensors class to make its scope explicit. Two interacting state machines are drawn for the two state attributes: The_Function_Status_Of and The_Calibration_Status_Of. The two initial states (i.e., Disabled and Uncalibrated) are shown as rounded rectangles

§ An object or class is said to export a feature (e.g., attribute type, constant attribute, message, exception, link) if the feature is declared in its interface. Exported features may be visible to other objects and classes, whereas non-exported features are nested in the implementation of the object and classes and protected by information hiding.

|| Timing probes are used to annotate diagrams (e.g., semantic nets, interaction diagrams, STDs, control flow diagrams) with timing information (e.g., requirements). The icon for a timing probe is connected to one or two arcs and documents the time when the arc *occurs* or the time difference between when the first and second arcs *occur*.

\# Attributes are protected if reads and writes to these attributes are guaranteed to execute without interruption. This protects the attribute values from corruption due to interleaved access in a concurrent environment.

drawn with thin dashed lines. The two final states are shown as rounded rectangles drawn with thick solid lines. The aggregate Functioning state is decomposed into the two atomic states Disabled and Enabled. The transition from Uncalibrated to Calibrated only occurs if a calibrated sensor is also in the Enabled state and the calibrate trigger operation executes. The exceptional transition from the aggregate Functioning state to the atomic Malfunctioning state occurs when the HARDWARE_FAILED exception handler of the update operation executes (i.e., the trigger is directly a result of the update operation and indirectly the result of a hardware sensor terminator).

STATE OPERATION TABLE

Because states determine the overall behavior of an object or class, states are often used to control the execution of operations. Some operations may commence execution only in certain prior states, and being in one of these allowed states becomes a *precondition* of the operation. Similarly, some operations may complete execution only in certain subsequent states and being in one of these allowed states becomes a *postcondition* of the operation. A *state operation table* (SOT) documents for each operation:

1. the prior simultaneous states required to exist before the operation can execute.
2. the associated exception to be raised by the precondition if the operation attempts to execute in an invalid prior state**
3. the associated valid subsequent states to be ensured by a postcondition upon completion of the correct execution of the operation

SOTs are used to (1) develop some of the preconditions and postconditions of the operations and (2) ensure consistency between the state model and the interaction model. Operations are annotated as either sequential or concurrent and as either modifier or preserver. Subsequent states that differ from initial states are emphasized in boldface. Table 1 documents the SOT for instances of the Calibrated_Sensors class.

INHERITANCE AND STATE MODELING

More research is required to adequately integrate state modeling and inheritance. If inheritance is used to capture generalization/specialization taxonomies, then (1) a derived class is a specialized kind of its base class (e.g., cars are *a kind of* vehicles) and (2) each instance of the derived class also can be considered to be an "instance" of the base class (e.g., My car *is a* vehicle). Just as the derived class is a specialization of the base class (possibly with additional characteristics), the state model of the derived class should in some sense be a specialization of the state model of the base class. For example, the state model of the derived class may contain additional state attributes or links resulting in the addition of subdiagrams to the STD of the base class. The derived class may introduce additional operations that require new transitions and triggers to be added to the state model of the base class. An atomic state of the base class may become an aggregate state of the derived class containing multiple related substates. If so, the new substates should be consistent with the original state of the base class (e.g., if the original state of the base class was not a final state, then neither should any of the associated new substates of the derived class; see Fig. 9). A derived class may also override (or even delete) resources inherited from its base class, especially if inheritance has been used to maximize code reuse rather than capture taxonomies. Multiple inheritance also may result in confusing interactions between state properties and operations that trigger transitions between states. For the preceding reasons, and until theory advances the understanding of the influence of inheritance on state modeling, ADM4 requires the state models of all concrete classes to be individually determined and documented.

DOCUMENTING THE ADM4 STATE MODEL

At the system (or software) level, the state model provides a dynamic view of the behavior of the system (or software) objects and classes in terms of their states, and the triggers that transition them from state to state. The state model at the system and software level consists of the following information (where appropriate), which should be captured in electronic form by a CASE tool:

- For each *class* with significant *state* behavior:
  1. A state transition diagram
  2. A state operation table
- For each *state*:
  1. Identifier
  2. Abstraction
  3. Condition characterizing the state, including state properties:
     - Attributes and values
     - Links
     - Component subobjects
  4. Responsibilities
  5. Requirements
  6. Superstates
  7. Substates (if aggregate)
  8. Transitions:
     - Incoming
     - Outgoing
  9. Behavior permitted:
     - Messages received
     - Messages sent

** A standard exception (e.g., Invalid_Subsequent_State) is usually raised by the postcondition if invalid subsequent states are detected. The SOT can be easily extended to list such exceptions if more flexibility is needed.

Table 2. Models and associated diagrams as a function of method.

Methodologists	Model	Diagram(s)
Firesmith's ADM4	State Model	State Transition Diagram (STD) State Operation Table (SOT)
Berard	*No separate model*	State Transition Diagram
Booch	*No separate model*	State Transition Diagram
Coad and Yourdon	*No separate model*	Object State Diagram
Coleman et al	Life-Cycle Model	Bubble Diagram and State Machine
Embley et al	Object-Behavior	State Net
Henderson-Sellers	**Missing**	**None**
Jacobson et al	*No separate model*	State Transition Graph
Lorenz	**Missing**	**None**
Martin and Odell	Activity Schema	State Transition Diagram*
Rumbaugh et al	Dynamic Model	State Diagram
Selic et al	*No separate model*	State Chart
Shlaer and Mellor	State Model	State Model
Wirfs-Brock et al	**Missing**	**None**

* Also known as a Fence Diagram or a State-Change Diagram.

Table 3. Definition of state.

	State Defined in terms of			
	Characteristics			Dynamic
Methodologists	Attributes	Links	Behavior	Classification
Firesmith's ADM4	Some	Some	Yes	No
Berard	Some	No	No	No
Booch	All	No	No	No
Coad and Yourdon	All	No	No	No
Coleman et al	?	?	?	?
Embley et al	No	No	Yes	No
Jacobson et al	All, Some	No	No	No
Martin and Odell	No	Yes	No	Yes
Rumbaugh et al	Yes	Yes	Yes	No
Selic et al	No	No	Yes	No
Shlaer and Mellor	No	No	Yes	No

Table 4. Categories of states by method.

	States			
Methodologists	Atomic	Aggregate	Initial	Final
Firesmith's ADM4	Yes	Yes	Initial	Final
Berard	Yes	**Missing**	**Missing**	**Missing**
Booch	Yes	**Missing**	Start	Stop
Coad and Yourdon	Yes	**Missing**	**Missing**	**Missing**
Coleman et al	Yes	**Missing**	Start	End
Embley et al	Yes	High-Level	Initial	Final
Jacobson et al	Yes	**Missing**	**Missing**	**Missing**
Martin and Odell	Yes	Yes	**Missing**	**Missing**
Rumbaugh et al	Yes	Yes	Initial	Final
Selic et al	Leaf	Composite	Yes	Yes
Shlaer and Mellor	Yes	**Missing**	Creation	Final

Table 5. Transitions, triggers, guards, and actions.

Methodologists	Transition	Trigger/Event	Guard	Action
Firesmith's ADM4	Transition	Trigger	Guard	Entry Operation
Berard	Transition	**Missing**	**Missing**	**Missing**
Booch	Transition	Event	**Missing**	**Action**
Coad and Yourdon	Transition	**Missing**	**Missing**	**Missing**
Coleman et al	Transition	Event	**Missing**	**Missing**
Embley et al	Transition	Trigger	**Missing**	**Action**
Jacobson et al	Transition	**Missing***	**Condition**	**Yes**
Martin and Odell	Transition	Event	Control Condition	Activity
Rumbaugh et al	Transition	Event	Guard	Action, Activity
Selic et al	Transition	Event	Guard	Action
Shlaer and Mellor	Transition	Event	**Missing**	Action

* Although Jacobson et al. do not explicitly discuss triggers or events, they do use several explicit icons for different events that may cause transitions.

- Operations executed
- Exceptions raised

- For each *guard condition*:
  1. Identifier
  2. Guard condition
  3. Transition
  4. Responsibilities
  5. Requirements
- For each *transition*:
  1. Identifier
  2. Trigger
  3. Trigger type (Normal vs. Exceptional)
  4. Trigger category (Operation vs. Terminator)
  5. Transition type (Leaves prior state vs. Remains in prior state)
  6. Concurrency type (Spontaneous vs. Non-spontaneous)
  7. Priority
  8. States:
     - Prior
     - Subsequent
  9. Guard condition
- For each state-related *timing probe*:
  1. Identifier
  2. Time including units
  3. Deadline type (hard, soft)
  4. Deadline category (minimum, average, maximum)
  5. Responsibilities
  6. Requirements

Table 6. Advanced forms of transitions.

Methodologists	Exceptional transition	Terminator	Spontaneous Transition	Remain in Prior State
Firesmith's ADM4	Exception	Terminator	Spontaneous	Yes
Berard	Missing	Missing	Missing	Missing
Booch	Missing	Missing	Missing	Missing
Coad and Yourdon	Missing	Missing	Missing	Missing
Coleman et al	Missing	Missing	Missing	Missing
Embley et al	Exception	Missing	Missing	Yes
Jacobson et al	Missing	Missing	Missing	Missing
Martin and Odell	Missing	Missing	Missing	Missing
Rumbaugh et al	Missing	Missing	Missing	Missing
Selic et al	Missing	Missing	Missing	Missing
Shlaer and Mellor	Missing	Missing	Missing	Missing

COMPARISON TO OTHER O-O METHODS

Because the properties of objects and classes can influence their overall behavior, most object-oriented development methods include some level of state modeling. Table 2 documents the state models and associated diagrams of the major object-oriented methodologists.

Some methods define state in terms of either some or all attributes encapsulated within the object or class. Some methods also include links in the definition. Yet other methods define state in terms of the behavior of objects and classes. Only one method (Martin and Odell[7]) defines state in terms of dynamic classification.†† Table 3 documents, on a method-by-method basis, the important concepts used in the definition of state.

Whereas all the methods compared include atomic states (although most do not have a separate term for it), many methods do not incorporate the Harel concept of aggregate states. Many methods also do not have separate terms (and icons) for initial and final states. Table 4 documents, on a method-by-method basis, the important types of states.

Whereas all the methods compared include transitions and most include triggers or events, the concept of a guard is incorporated in only half the methods. The concept of an action as a result of a transition is included in ADM4 only in the form of entry operations associated with state (as opposed to being associated with the transition). Table 5 documents, on a method-by-method basis, support for the concepts of transition, trigger or event, guard, and action.

Few methods support advanced topics such as exceptional transitions, transitions caused by terminators that violate object encapsulation, spontaneous transitions, or transitions that allow the object to remain in its prior state while also transitioning to a new state. Table 6 documents method support for these advanced forms of transitions.

CONCLUSION

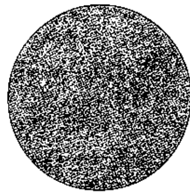
ADM4 incorporates a very powerful‡‡ state modeling approach in which important state modeling concepts are defined in terms of object-oriented concepts. Thus, state is defined in terms of properties that qualitatively determine object and class behavior. This makes ADM4 state modeling inherently object oriented and well integrated with the remainder of the method. ADM4 thus provides power while retaining compatibility with the object paradigm. ■

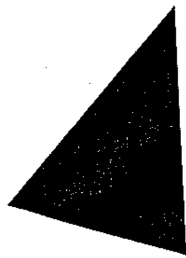
References

1. Firesmith, D.G. Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach, John Wiley, New York, 1993.
2. Booth, T.L. Sequential Machines and Automata Theory, John Wiley, New York, 1967.
3. Harel, D. Statecharts: A visual formalism for complex systems, Science of Computer Programming 8(3):231–274, 1987.
4. Embley, D.W., B.D. Kurtz, and S.N. Woodfield. Object-Oriented Systems Analysis: A Model-Driven Approach, Prentice Hall, Englewood Cliffs, NJ, 1992.
5. Selic, B., G. Gullekson, and P.T. Ward. Real-Time Object-Oriented Modeling, John Wiley, New York, 1993.
6. Hopcroft, J. and J. Ullman. Introduction to Automata Theory, Addison-Wesley, Reading, MA, 1979.
7. Martin, J. and J.J. Odell. Object-Oriented Analysis and Design, Prentice Hall, Englewood Cliffs, NJ, 1992.
8. J Palmer. A state of state confusion, Object Magazine 3(2):34, 1993.
9. Shlaer, S. and S.J. Mellor. Object-Oriented Systems Analysis: Modeling the World in Data, Prentice Hall, Englewood Cliffs, NJ, 1988.
10. Shlaer, S. and S.J. Mellor. Object Lifecycles, Modeling the World in Data, Prentice Hall, Englewood Cliffs, NJ, 1992.
11. Berard, E.V. Essays on Object-Oriented Software Engineering, Vol. I, Prentice Hall, Englewood Cliffs, NJ, 1993.
12. Booch, G. Object-Oriented Design with Applications, Benjamin/Cummings, Menlo Park, CA, 1991.
13. Coad, P. and E. Yourdon. Object-Oriented Analysis, 2nd Ed., Prentice Hall, Englewood Cliffs, NJ, 1990.
14. Coad, P. and E. Yourdon. Object-Oriented Design, Prentice Hall, Englewood Cliffs, NJ, 1991.
15. B Henderson-Sellers. A Book of Object-Oriented Knowledge: Object-Oriented Analysis, Design, and Implementation: A New Approach to Software Engineering, Prentice Hall, Paramatta, New South Wales, Australia, 1992.
16. Jacobson, I. et al. Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Wokingham, UK, 1992.
17. Lorenz, M. Object-Oriented Software Development: A Practical Guide, Prentice Hall, Englewood Cliffs, NJ, 1993.
18. Rumbaugh, J. et al. Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, 1991.
19. Wirfs-Brock, R., B. Wilkerson, and L. Wiener. Designing Object-Oriented Software, Prentice Hall, Englewood Cliffs, NJ, 1990.

†† See "A state of state confusion" by John Palmer[8] for arguments against using dynamic classification for state transitions.

‡‡ ADM is based on the philosophy that it is better to have a tool and not need it (or only need it rarely) than to need a tool and not have it. ADM therefore tends to be a superset of other methods, and it may be tailored down for small projects. Unlike some methods, which are strongly based on state modeling (e.g., Shlaer and Mellor[9,10]), ADM does not require state modeling for all objects and classes. However, ADM does provide a very powerful model when needed.





JOURNAL OF
OBJECT-ORIENTED programming

January 1995
Vol. 7, No. 8

Editorial 4

Guest Editorial 6
Systems, objects, and details
Jon Cartiant

Guest Column 7
Patterns: PLoP, PLoP, fizz, fizz
Robert Martin

Analysis & Design 14
Approaches to finite-state machine modeling
James Odell

Modeling & Design 21
OMT: The object model
James Rumbaugh

A Deeper Look... 28
...at translating actions
Stephen J. Mellor

C++ 66
Introduction to iterator adaptors
Andrew Koenig

Smalltalk 69
Communicating reusable designs via design patterns
Wilf LaLonde & John Pugh

Eiffel 72
Lessons learned in a first Eiffel project
Robert Howard & Shahzad Pervez

Product Review 75
C++/Views
Lewis J. Pinson

Book Review 76
DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE
Sanjiv Gossain

Book Review 90
OBJECT-ORIENTED PROGRAMMING WITH PROTOTYPES
Steven C. Bilow

Ad Index 88

Recruitment 94

Product News 96

Features

Object-oriented models of functionally integrated computer systems 32
Jens Kaasbøll

Many people work with more than one computer program and experience problems related to functional integration. O-O methods are well suited for developing components useful in several applications. A framework for capturing functional integration in O-O analysis and design is proposed. Three O-O approaches are examined in relation to functional integration, but none cover all relevant aspects.

Behavorial specifications in object-oriented programming 41
Douglas Skuce & Ali Mili

The authors show how a relation-based model can formally specify the behaviour of objects in an object-oriented programming context. They discuss how this model can be used to represent compound objects, and how it represents inheritance. This model is illustrated with the familiar ATM specification.

An object-oriented intermediate code representation for the development of parallelization tools 50
Mark R. Gilder and Mukkai S. Krishnamoorthy

Source code portability is difficult, particularly when mapping to parallel architectures. Using an intermediate code representation capable of efficiently representing both the high-level source code constructs and low-level information contained therein greatly simplifies the design of parallel analysis tools.

Object-oriented state modeling using ADM4 57
Donald G. Firesmith

This article summarizes the state modeling approach of the ASTS Development Method (ADM). The author defines basic concepts of state modeling, introduces ADM's state transition diagrams and operation tables, presents two examples, and compares ADM's state model to those of other methods.

The JOURNAL OF OBJECT-ORIENTED PROGRAMMING (ISSN #0896-8438) is published nine times a year, monthly except for Mar/Apr, Jul/Aug, and Nov/Dec by SIGS Publications Inc., 71 West 23rd Street, 3rd floor, New York, New York 10010. Please direct advertising inquiries to this address. Second class postage paid at New York, New York, and additional mailing offices. POSTMASTER: Send address changes to JOOP, P.O. Box 2030, Langhorne, PA 19047. Inquiries and new subscription orders should also be sent to that address. Annual subscription rates for the U.S. are $199 for institutions, $69 for individuals. All foreign orders must be prepaid in U.S. funds drawn on a U.S. bank. Canadian & Mexican orders add $25 per year and non-North American orders add $40 per air service. For service on current subscriptions, call 215.785.5996, fax 215.785.6073, e-mail p00976@psilink.com.

© Copyright 1995 SIGS Publications Inc. All rights reserved. Reproduction of this material by electronic transmission, Xerox, or any other method will be treated as a willful violation of the US Copyright law and is flatly prohibited. Material may be reproduced with express permission from the publisher. Statements of opinion and fact are made on the responsibility of the authors alone and do not imply an opinion on the part of SIGS PUBLICATIONS INC. or the editorial staff. All trademarks are the property of their respective owners.

Manuscripts under review should be typed double spaced (in triplicate) and accompanied by an electronic file in TEXT format. Editorial correspondence and Product News information should be sent to the Editor, Dr. Richard S. Wiener, 135 Rugely Court, Colorado Springs, CO 80906, 719.579.9616 (voice & fax).

Printed in the USA. Canada Post International Publications Mail Product Sales Agreement No. 290343.