A Note on Inheritance and State Machines

John D. McGregor[1]
Douglas M. Dyer[2]

Abstract

The inheritance relationships in an object-oriented program have been shown to be a natural structure for facilitating reuse in a number of areas such as test cases. This note reports on a technique for incrementally building the state machine for a class from the state machines of its base classes. Using a style of state diagram similar to objectcharts provides a graphical means of displaying the resulting state machine. This incremental technique allows more complex machines to be built with less effort.

1 Introduction

This is a report of intermediate results from work undertaken to extend previously reported testing techniques for object-oriented software systems[4]. A goal of this research was to better understand the relationship between a state machine representation of a class and similar representations for its subclasses. The results are being used to develop techniques that support the reuse of state machine information along the hierarchical structure constructed by the inheritance relationships among classes.

Although the dynamics of most objects can be represented by a set of states and the transitions between those states, most objects do not constitute formal finite automata. The techniques used in this design methodology do not require the formality of the automata in order to benefit from the information provided in a state machine representation. The discussion will include the formality that is possible, but without requiring restrictions that are not practical for a majority of objects.

For purposes of this research, as well as most of our work, we restrict the inheritance relationship to the so-called *strict* inheritance model that is increasingly being accepted as the model that supports a subtyping relationship. McGregor and Sykes[8] consider the inheritance relationship between two classes as a mapping that carries out certain transforms on attributes as they are inherited from one class into another. Each object-oriented language has its own set of transforms that may be applied.

The model of inheritance used in this research does not allow a mapping that either fails to map a public method to the new class (cancellation) or that changes the visibility of a public method. This restriction of the inheritance mapping supports inferencing about the behavior of the resulting classes. This inferencing has already been the basis of an incremental testing technique[4, 5].

[1]Object Technology Group; Department of Computer Science; Clemson University; Clemson, SC 29634; johnmc@cs.clemson.edu
[2]Software Technology, Inc.; 5904 Richmond Hwy.; Alexandria, VA 22303

Figure 1: The set of value vectors

We also require that all data attributes be hidden and only accessible through the methods in the public interface of the class.

A second issue that was addressed in the research was the possibility of multiple parents for a new class. A number of discussions about the viability of multiple inheritance have appeared in the literature[1]. Our position is that, with the appropriate assumptions, multiple inheritance is a useful tool. One objective of this part of the research was to provide a basis for postulating design rules that present a reasonable approach to the use of multiple inheritance.

In the next section we provide background about the use of state machines and their diagrams for representing classes, in section 3 we discuss the main ideas of inheriting the state machine representation, in section 4 we add the complications of multiple inheritance, and we conclude by considering the impact of these results on testing techniques.

2 Background

A major thrust of this research assumes that the *state* of an object and its set of possible states are a useful means of representing portions of the behavior of object-oriented systems. We will first explore the concept of state of an object and then consider the styles of diagrams that are used to present this information.

2.1 States

A "state" in the usual sense, may be taken as a vector of values containing one entry for each of the data attributes of the object. We will denote one of these value vectors as $v =< a_1, a_2, \ldots, a_n >$ where each a_i is the current

Figure 2: A partitioning of the value vectors

value of one attribute of the object. The **state space**, S, of an object is the cartesian product of the state spaces for each of the data attributes, see Figure 1. This definition of state is not very useful from a modeling perspective since any change in the value of any attribute results in a "different" state. This leads to a potentially-infinite set of states that is worthless in guiding modeling or testing activities. Therefore, we will consider a *design-level* state, s, to be some set of value vectors, $\{v_1, v_2, \ldots, v_n\}$, such that vectors in one set are clearly differentiated from those in another set by some classification rule, see Figure 2. For example, typically the value of a single integer variable, X, changing from 1 to 2 to 3 and on is of only local interest and calling each a separate state provides no additional information; on the other hand, if X changes from being positive to zero or changes from zero to negative, these changes are often viewed as significant. A set of meaningful states for X might then be positive, zero, and negative.

A second requirement, for the definition of a design-level state, is that the state must be based on the observable behavior of the object. An object is designed to conceal details of implementation and to present a behavioral interface to potential clients. The set of states is defined in terms of the values returned by the public methods of the object. This eliminates the need to consider attributes that have housekeeping duties and has the effect of reducing the number of v_i's that need to be considered. For example, in an array-based implementation of a stack, there maybe from 0 to n elements on the stack as indicated by the top-of-stack pointer and; therefore, potentially n states of the stack. From a design perspective we are usually only interested in whether the stack is empty, full, or someplace inbetween. This information can be provided publicly through two operators: EMPTY() and FULL(). Rather than considering n states we need only to consider three.

Based on these requirements, our working definition of *design-level* state is:

A set of vectors of observable values that share some behavioral attribute of interest.

Except where noted, we will refer to design-level states simply as states for the remainder of the paper.

We will further assume that the specification of an object includes a state space, S, and an event space, E, which together define the behavior of the object. A transition space, T, assists in modeling the dynamics of how objects move from one state to another.

The state space, S, of a class is a "partitioning" of the value vectors, v_i, based on rules of classification stated as part of the design criteria. The partitioning results in a set of equivalence classes that define the states of the object and meets the criteria that if v is a value vector for the object then this implies that $v \in s_i$; for some s_i. That is, S includes every possible v for the component. Further we assume that, while the grouping of v's into s_i's is by no means unique, for a given decomposition, the s_i's are disjoint. So S forms an exhaustive, mutually exclusive partitioning for the set of value vectors that represent the behaviors of the object.

The event space, E, of an object comprises those messages that display the meaningful behavior of the object. In most object-oriented languages, this corresponds to a subset of the methods defined on the class, assuming that all of the data attributes are protected from manipulation from outside the class. Each of the public methods, that modify the value of any attribute, implements an event.

The transition space, T, of a component is a set of triples of the form

(<input vector>, event, <output vector>)

The input vector for a transition includes the current state of the component and any input parameters required by the event. The output vector includes the resulting state of the component as well as any values returned by the event. The elements of the transition space, t_i, correspond to the transitions defined by the state machine. A single event can, and often does, participate in several transitions.

2.2 State Diagrams

A number of object-oriented development methodologies utilize a state representation to capture the dynamics of class instances, including the technique proposed by Rumbaugh et al[9] and that of Shlaer and Mellor[10]. Rumbaugh's approach includes a **dynamic model** for a class that captures the changes in state undergone by instances in response to messages.

Several styles of state diagrams have been reported in the literature[2, 6, 9]. The style that we will use is a blend of several of these approaches. In particular it will use two important features provided by the **objectchart** style presented

Figure 3: Objectchart for the graphics display unit

by Coleman et al[2]. Objectcharts are a derivative of the statechart concept originated by Harel[3]. Objectcharts introduce the concepts of hierarchy and concurrency into the state representation.

The concept of hierarchy is present in the form of substates. That is, one state may be divided into several independent subdivisions, only one of which describes the state of the system at a time. This hierarchical scheme supports varying levels of specificity depending upon the need. For example, a graphics display unit may be on or off. This set of states may be represented as seen in Figure 3(a). We might further be interested in whether a visual image is being displayed or not. Obviously this question is only of interest if the display unit is on. These substates, see Figure 3(b), of the original state "display unit on" provide additional detail about the encompassing state.

An added piece of notation supports the concept of concurrency. Many display units have a hardware switch to select either a horizontal or vertical picture display. This hardware switch may be repositioned whether or not the display is currently on. Figure 3(c) illustrates the objectchart notation, the

vertical dotted line, for this relationship.

These two constructs provide the additional power needed to model object-oriented systems.

2.3 Strict inheritance

The inheritance relation has been used in software development in two very different ways. The first is termed "inheritance of implementation." The objective from this perspective is to build a new class by "borrowing" some implementation details from an existing class. In the second approach, termed "inheritance of specification", the new class is designed by including the specification of an existing class as a subset of the specification of the new class.

Our definition of "strict" inheritance adopts the specification approach by assuming that each subclass contains the public interface of its parents as a subset of its public interface. However, the requirements of strict inheritance go beyond the textual matching of method names; the **intention** of those methods must remain constant. This can be more formally expressed by stating a **method invariant** for each method that is inherited by the new class:

> *The pre-conditions on a method in the subclass may only be weakened relative to the pre-conditions on the same method in the base class. The post-conditions on a method in the subclass may only be strengthened relative to the post-conditions on the same method in the base class.*

This leads to the statement of a **class invariant** as:

> *The specification of a class must include, as a subset, the specification of each of its base classes. Additionally, the method invariant must hold for every inherited method in the subclass. By implication, every class has in its specification the specification of the root of each inheritance structure within which it participates.*

These two requirements produce an inheritance hierarchy in which the subclasses are subtype compatible with their parents and polymorphic substitution of subclasses for parents is possible. They also produce a structure about which logical inferences can be made.

3 Inheriting State Machines

The creation of a state representation for a software component is not as exact a process as is the process for circuit design, for example. Two designers may derive very different state machines for the same component depending upon

the characteristics that are considered sufficently important to use in aggregating value vectors. It is also the case that determining those characteristics is a difficult task. One of the goals of this research was to consider how the inheritance mapping might be used to construct state representations for a class from the state representations of its parents.

We begin by assuming that it is possible to represent the behavior of a class with a state machine. In general, it is not possible to assume that the state machine of every class represents a formal automaton, although in practice we can come very close. Our techniques do not require such a level of rigor. The value vectors for a class A contain n data values, where n is the number of distinct methods in A that exhibit behavior. We will represent such a vector as follows:

$$v_i =< r_{A_1}, r_{A_2}, \ldots, r_{A_n} > \tag{1}$$

where r_{A_i} is the return value of the ith method of class A. Therefore, a state of class A is a set of value vectors:

$$s_{A_i} = \{v_{A_1}, \ldots, v_{A_m}\}. \tag{2}$$

where m is the number of value vectors that are included in the state. There is some selection rule for the set that determines which value vectors are members of that state. For example, $r_{A_2} = TRUE$ for every v_{A_i} in s_{A_i}, indicates that this state is the set of all vectors whose second component is true. For a stack object, the true value might be the return value of the EMPTY() method and the state s_{A_i} is actually the empty state for the stack.

3.1 Implications of Strict Inheritance Assumption

The assumption of the strict inheritance model allows us to derive a set of implications about the state machine for the new class formed from a set of parent classes. Figure 4 summarizes those implications.

The box labeled class A contains a state diagram for this root class. The state diagram for class B illustrates that a transition may be added to the state diagram for class A to construct the diagram for class B. In class C, one of the states from class A is decomposed into two substates. A method from class A would still see the single state while a method from class C could take advantage of additional information to discriminate between the two substates. In class D a set of attributes are added to those inherited from class A resulting in a set of concurrent states. Class E is intended to indicate that an inheriting class may modify a state by changing the interpretation even though the diagram does not change significantly. In class F, one of the state transitions is modified. This is most often accomplished by overriding an inherited method.

The following is a discussion of a few of the most significant implications.

Implication 1: *A child class can not delete a state of any of its parent classes.* Given the assumption of a strict inheritance model, the states of the

Figure 4: Implications of the strict inheritance model.

parent can be seen in the child. For example, if the value vectors for class A are as given in equation 1, then the value vectors for class B, where B is a subclass of A, would be given by:

$$v_{B_i} =< r_{A_1}, \ldots, r_{A_n}, r_{B_1}, \ldots, r_{B_m} > . \tag{3}$$

where m is the number of observable data values introduced in B that were not in A. A simple projection transformation could be defined that maps the v_B's onto the v_A's. The result is a many-to-one mapping that associates a set of value vectors in B with a value vector of A.

As illustrated by the shaded area in class E in Figure 4, it is possible to modify a state in the sense that the data representation may be modified provided that the "observable behavior" of the methods does not change. This corresponds to the requirement that the method invariant hold for every method in the class stated in the class invariant.

Implication 2: *Any new state introduced in a child class is wholly contained in an existing state of one of the parent classes.* There may be significant design factors that lead to a further partitioning of the v_B's. In that case one of two situations is possible. First the additional partitioning does not cross the boundaries of the original partitioning of the v_A's or the partitioning of the v_B's is based solely on values that were not used in the original partitioning. In the former case, illustrated in Figure 5, the result is substates in B of the original states in A. The latter case results in a set of states that are independent of and

Figure 5: Substates of the original states.

disjoint from the original set of states in A. In this case an instance of B can be thought of as being in one state from each set concurrently.

A third scenario, an interaction among the data attributes used in the original partitioning and the attributes to be used in the new partitioning that results in a state that might overlap two existing states, is not possible under the strict inheritance assumption which is a behavior-preserving mapping and based on the definition of substates. Consider a new state, s_5, that includes portions of states s_3 and s_4. By our original assumptions, s_5 is disjoint from any other state; however, the simple projection function illustrates that any value vector is either a member of s_3 or a member of s_4 but not both.

Implication 3: *A child class may not delete a transition from the state machine of one of its parent classes.* If a transition were deleted then the method that implements that transition must have been modified to loosen its post-conditions, a violation of the method invariant. However, as shown in class F in Figure 4, it is possible for a class to override an inherited method and change its implementation provided that its post-conditions remain the same as or more restricted than those of the overriden method.

Class B in Figure 4 illustrates that a transition may be added to the machine. This will often be a "convenience" function that combines the actions of existing functions and bypasses a series of intermediate states on its way to the desired state.

Figure 6: A simple example

3.2 Examples

As previously stated, there are many ways to build a state machine for a given class; therefore, the following examples do not claim to be the best state representation for the given class. The classes in the examples have been simplified to result in machines that are easy to build and understand.

In this first example the derived class specializes the intention of the base class. In this example we begin with a Window class, Figure 6(a), that has two states: mapped and unmapped. A window may be mapped to the display screen or it is hidden. A method in the class would provide the return value that is used for the value vector. This method can also be used by a window manager to eliminate the need for operating on windows that are not displayed.

The subclass, ScrollableWindow, adds a set of data attributes to the original class. These attributes describe whether the window has been refreshed since it was last scrolled. These states are not independent of the states of the original window. When a window is not mapped to the display it can not be scrolled and thus does not need to be refreshed. These new states are only of interest if the window is mapped to the display. Figure 6 shows that this relationship can be represented as substates of the mapped state.

In this second example we begin again with the simple Window class from the previous example. The states for this class are shown in Figure 7. The second class, EditWindow, is a window in which a file can be edited. In order to

Figure 7: An example of concurrent states

be edited, a file must first be opened. This leads to the states: open and closed. A file can be open whether or not the window is mapped to the display or not. Since the states for the file are independent of the states for the window, this results in the state machine illustrated in Figure 7(b). The vertical dashed line indicates two concurrent sets of states. The object is considered to be in one state from each set at any given moment.

4 Multiple Inheritance

4.1 Types of Multiple Inheritance

The use of multiple parents as the basis for a new class adds complexity at both the implementation and design levels. At the implementation level, some mechanism must exist to resolve any conflicts that exist as two name spaces are merged into one. This situation is further exacerbated if some of the parent classes have a common ancestor resulting in "repeated" inheritance of some attributes. The amount of automation available to recognize and mitigate these problems varies with the implementation language, but usually includes compiler support for identifying conflicts.

At the design level, the requirement for appropriate use of multiple inheritance has been set forth by Carré and Geib[1] that the classes being used as base classes must represent independent concepts. The independence of the

concepts implies disjoint representations (This does not preclude name clashes at a textual level, but that possibility will be ignored in this discussion).

The representations of the base classes define state machines for their original classes. The independence of the concepts and the disjointness of the representations imply that the state machines of the separate classes do not have any overlap. The class that results from the inheritance of the multiple base classes should have a state machine that contains these machines simultaneously. The derived class then is in a state of each of its base classes concurrently.

4.2 Examples

Let's consider the example illustrated by the inheritance hierarchy shown in the center of Figure 8. In this example existing constraint and graphical object classes are used to derive a class of constraints that can be drawn on a graphics display. The state diagrams, shown in Figure 8 for the two root classes are reasonably simple. A constraint is either in the state of being satisfied or that of being unsatisfied. A graphical object is either in the state of having been mapped to—displayed on—the display or not. Instances of the new class inherit all of the attributes of their parents and so must accomodate all of the states of the parents. A graphical constraint can be drawn on the display whether or not it is in the satisfied state. Likewise, it can be satisfied whether it is currently being displayed or not. This results in a representation in which the objects are in two states concurrently as illustrated by the bottom diagram in Figure 8.

Consider a second example involving repeated inheritance of the state machine of the common ancestor and the impact on a new subclass. In this example, a bounded priority queue is created by use of the classes in the inheritance hierarchy shown in Figure 9. Figure 10(a) illustrates the simple state machine for the Queue class. In this figure we introduce a new notation. The vertical box, containing the notation T(rue) F(alse), reduces the complexity of the diagram slightly and indicates the condition upon which the two paths are based.

The state machines for BoundedQueue and PriorityQueue are constructed from the machine shown for Queue. Adding the designation of Bounded to the Queue introduces the possibility of the queue being full. This new state is accommodated by decomposing the NotEmpty state into two states, PartialFull and Full. These are substates of the original NotEmpty state. Thus, an instance of BoundedQueue that is in the PartialFull state can also be thought of as being in the NotEmpty state and will behave as an instance of Queue in the same state.

The addition of the Priority designation to Queue results in the addition of the possibilities of either a new maximum value in the queue or the queue being reordered to accommodate a new value that is not a new maximum. These possibilities are represented in Figure 11 as substates of the NotEmpty state. Once again an instance of PriorityQueue that is in the NewTopElement state will also behave as if it were in the NotEmpty state.

Figure 8: State machines using multiple inheritance.

Figure 10: State diagrams for Queue and BoundedQueue

Figure 12: BoundedPriorityQueue

The final class in this hierarchy is BoundedPriorityQueue, constructed by inheriting from BoundedQueue and PriorityQueue. The machine for this class can be constructed by considering that in each of the parent classes' machines the NotEmpty state was subdivided into a pair of substates. The concept of being Bounded is independent of the concept of ordering based on a Priority. Thus, the queue can be the Full state and be in either the NewTopElement or the NewShape state. The same applies to PartialFull. This is represented in Figure 12 as pairs of concurrent states.

The state machine in Figure 12 is reasonably complex, but, by building up the machines for each class beginning with the simple root class, each new machine can be developed by reasoning about the ones that have already been built. The incremental approach to constructing these machines results in a decreased amount of effort required for building each machine and the ability to build more complex machines.

5 Conclusions

A central thesis of this paper has been that the behavior of a class can be modeled as a finite state machine. This state machine view of a class is useful for such operations as systematically developing the specification test suite for the class[7]. State machines are difficult to construct in many cases; therefore, a technique to incrementally construct the state machine for a class was investigated.

The goal of this paper was to present such a construction technique. In order to support such an algorithmic approach to state machine construction, a restriction on the use of the inheritance relationship, termed strict inheritance, was defined. The use of this relationship results in a new class that can be treated as a subtype of the ancestor classes.

Two techniques were used in the algorithm. First, a state from a base class could be decomposed into two or more substates in the derived class. The second technique was to add a new set of states that are in parallel to those that existed in the base classes. These techniques together with the restricted inheritance mechanism provide sufficient power to incrementally modify the state machines from ancestor classes to form the machine that describes the derived class. A series of implications were derived that can be used to reason about the possible forms that the new machine might take.

The technique reported in this note provides support for our research on the specification-based testing of classes. We are developing an algorithm for developing test cases and establishing an adequacy criteria for determining when a sufficient set of cases has been constructed. The algorithm relies on the availability of the state machine representation of a class. The work reported here facilitates the construction of that representation and thus decreases the effort required to build the test suite.

Acknowledgement: The authors would like to thank the students of the Object-Oriented Research Seminar at Clemson University and Dr. Tim Korson for their suggestions in the preparation of this paper.

References

[1] Bernard Carré and Jean-Marc Geib. The point of view notion for multiple inheritance. In *Proceedings of OOPSLA '90*. ACM, 1990.

[2] D. Coleman, F. Hayes, and S. Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9 — 18, 1992.

[3] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231 — 274, 1987.

[4] Mary Jean Harrold and John D. McGregor. Hierarchical incremental testing. Technical Report TR91-111, Department of Computer Science, Clemson University, 1991.

[5] Mary Jean Harrold, John D. McGregor, and Kevin Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceedings of the 14th International Conference on Software Engineering*, 1992.

[6] G. Kappel and M. Schrefl. Using an object-oriented diagram technique for the design of information systems. In *Proceedings of the International Working Conference on Dynamic Modelling of Information Systems*, 1990.

[7] John D. McGregor and Douglas M. Dyer. Selecting specification-based test cases for object-oriented systems. Technical Report To appear, Clemson University.

[8] John D. McGregor and David A. Sykes. *Object-Oriented Software Development: Engineering Software for Reuse*. Van Nostrand Reinhold, 1992.

[9] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[10] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Analysis: Modeling the World with Data*. Prentice-Hall, Englewood Cliffs, NJ, 1989.