A Framework for Multiple, Concurrent Graphical Representation

Kevin Brophy         Michael D. Maggio
Nixdorf Computer Engineering Corporation
52 Third Avenue
Burlington, MA 01803

ABSTRACT

This paper presents an object-oriented framework that supports multiple concurrent representations of objects, including their semantics and their interrelationships with other objects. The framework is based primarily on two object classes, the Building-Level Object Class (BLOC) and the Representation Object Class (ROC). The two classes, BLOC and ROC, along with a method for defining relationships between these classes, provides support for multiple concurrent representations.

Since the introduction of powerful, graphics-based workstations, various efforts have been made to provide a natural way for representing tasks, models, and programs. For example, early software development systems attempted to apply the use of graphics to traditional software engineering problems. These systems devoted a large amount of effort to illustrating the program structure and output. However, the graphics were primarily used as presentations for static structures defined within the systems. The next genre of systems used graphics as a means of manipulating a program using various representations of the programs. These systems allowed a user to graphically interact with various components of a program, e.g., data structures, control flow, and symbol table definitions [1][2][3][4][5]. By providing a method for creating visual representations for the concepts or models that describe a problem, along with a means of directly executing the representation, the most recent systems have attempted to use graphical representation as the solution [6][7]. These systems adopted a methodology for associating data and semantics with objects and their representations within the model.

In order to show the motivation and support for the object-oriented framework, this paper presents and synthesizes observations and results from related research and commercial systems. A description of the proposed framework will be given, including a model for object definition and representation. Finally, to illustrate the use of the framework, a detailed example will be presented.

Background

In the software engineering field, various graphics-based software development systems have shown that information is best captured and manipulated through the use of several different, yet consistent, graphical views of that information. For example, a program and its components can be created and modified using text, a flow graph, a parse tree, symbol table and type definitions [1][3][5]. In order to achieve the consistency between the various graphical views, these systems generally use a fixed underlying canonical form for the data they are manipulating. Given a more general underlying model, a software development environment can represent multiple aspects of the software lifecycle: specifications, design, implementation, and project management [8]. The use of multiple, consistent views within such an environment becomes necessary in order to capture the diversity and complexity of this information. However, the views constructed in these systems are limited to the structure of the underlying model and are therefore not generally extendible or reusable.
The task of constructing graphics-based systems and supporting multiple views is difficult and time consuming. Object-oriented programming techniques have proven to be extremely useful and practical for providing efficient, sophisticated graphical interfaces and representations, that are, by nature, both extendible and reusable. For example, an application framework, a set of interconnected objects that model the basic structure of the user interface and its components, allows the developer to concentrate on the application rather than the user interface aspects [9]. Furthermore, given the ability to encapsulate data and procedures within objects, it becomes possible to associate semantics directly with graphical representations, allowing interaction at a higher level of abstraction. For example, an application may consider Time as a 3-tuple, (hours:minutes:seconds), whereas the interface can represent this information as an analog or digital clock. The drawing information for these representations is contained within the interface objects, not in the application [lo]. While this method promotes separation of representation knowledge from application knowledge, it is also possible to define application knowledge in terms of its representation. In fact, a graphical environment can be used to interactively create certain applications defined in terms of their interface representations and associated semantics [7].

Recent research environments have suggested that objects can represent not only data, but also computation, in a manner similar to COMMON LISP [11], where a symbol has both a value and a function binding. For example, an object can be evaluated by examining its associated semantics and taking appropriate actions. Furthermore, by defining relationships between these objects, it is possible to establish an executable conceptual model [6]. Finally, by combining these conceptual models it is possible to create a larger more abstract object. For example, it has been shown that commitments in a negotiation can be modeled with protocols, defined using a Finite State Automata (FSA), and terms, defined using an Entity Relationship Attribute data model (ERA) [12]. By defining conceptual models of an FSA and an ERA, it is then possible to define a commitment in terms of those models.

The results and observations of the systems described above suggest a basic strategy and a set of requirements for providing graphical representation. The strategy for defining objects within the framework is based on the concept of separating the functional model from the set of representation models. The requirements include: multiple, consistent representations of objects; the association of representation semantics with objects; and a flexible mechanism for defining object interrelationship. It is the combination of this strategy and these three criteria that lead to a Representation Framework.

The Representation Framework

The representation framework, using an object-oriented paradigm, is based on two primary object classes, the Building-Level Object Class (BLOC), which defines the functional model, and the Representation Object Class (ROC), which defines the set of representation models. These two classes are initially derived from a root object class containing the fields Name, Script, and Dependents. The Name field is a text identifier for the object. The Script is a set of methods which define how the object responds to specific events, thus defining the semantics of the object. The Dependents are a set of objects that have some type of dependency on the given object. For instance, in the commitment negotiation example, if an FSA object is a shared protocol among several commitment objects, then the commitments are dependents of the FSA object. The two classes, BLOC and ROC, combined with a methodology for defining relationships between them, defines the Representation Framework.

BLOC

The Building-Level Object Class contains the fields Name Script, Dependents, Scope, and Reps, as illustrated in Figure 1. The Scope is the object that binds the given object to a functional model; i.e., the Scope captures the hierarchical structure of the functional model. For example, in a functional model of a finite state automata, a state object's scope would be the FSA Object in which it is defined. Whenever an object's script cannot respond to some message, the message is passed up to the Scope object. The Reps field contains the set of representation objects.
associated with the given object. Reps are a special case of dependents, but have been separated for clarity.

Figure 1. Building-Level Object Class

ROC

The Representation Object Class contains the fields Name, Script, Dependents, Item, MetaRep, SubReps, Location, and Display, as illustrated in Figure 2. Item is the building-level object for which this object is a representation. MetaRep, SubRep, Location, and Display define a presentation for the item. The MetaRep is the representation object (rep) in which the given object appears. SubReps are those reps appearing within the given object. Location is a description of where the given object is placed within the MetaRep. Finally, the Display is a description of how the given object is drawn or constructed.

Figure 2. Representation Object Class

Example: An FSA

In order to demonstrate the use of the representation framework, an example defining

multiple representations of a finite state automata is presented. In this example, instances of the BLOC will be referred to as objects and instances of the ROC will be called reps.

An FSA can be defined as a set of allowed tokens, states, and transitions between the states, as well as the concept of a current state, which will be used during the execution of the FSA. As shown in Figure 3, the FSA object class maps directly to this definition. A portion of the script for FSA objects describing the actions that take place when the FSA receives a RUN message is also shown. These actions will be explained in a later section.

Figure 3. FSA Object Class

Two classic representations for an FSA are the state-transition diagram and the state-transition table. Figure 4 shows two representation classes that are defined and associated with the FSA object class. The first class, FSArep1, is simply the drawing area for a state-transition diagram. Instances of this class contain a window displaying the FSA's name, and a menu for manipulating and executing the FSA. Instances of the second class, FSArep2, display the FSA's name, as well as an empty table for a state-transition table.

Figure 5 shows the State object class and its two associated representations classes. An instance of the first rep class, STATErep1, is displayed as a node containing the state's name, as is typically found in a state-transition diagram. An instance of the second rep class, STATErep2, is displayed simply as the state's name, which will be placed in the state-transition table. The portion of the
script shown for the State objects describes the actions taken when a State receives an ACTIVATE or DEACTIVATE message.

A transition can be defined in terms of a 3-tuple \((\text{FromState, trigger, ToState})\) and an action to take place on the transition. As shown in Figure 6, the Transition object class also has two related representation classes, \(\text{TRANrep1}\) and \(\text{TRANrep2}\). An instance of \(\text{TRANrep1}\) displays a transition as an arc with the trigger name as a label. An instance of \(\text{TRANrep2}\) displays the name of the To-State. The portion of the script shown for Transition objects describes the actions to take place when a Transition receives a RUN message.

The functionality of an FSA can be completely defined in terms of the FSA object class, the State object class, and the Transition object class. By associating the appropriate representations with instances of these classes, an FSA can then be viewed and manipulated graphically.

Representing an FSA

This section will describe a simple FSA called \(\text{SimpleFSA}\). \(\text{SimpleFSA}\) has the States A, B, and C and the Transitions (A,b,B) and (B,a,A), and (B,c,C). Using the two types of representations for an FSA, State, and Transition objects, multiple, consistent views of SimpleFSA are produced. Figure 7 shows the state-transition diagram and state-transition table for SimpleFSA. The diagram shows each view of SimpleFSA as the composition of instances of \(\text{FSArep1}\), \(\text{STATErep1}\), and \(\text{TRANrep1}\) to define the state-transition diagram and \(\text{FSArep2}\), \(\text{STATErep2}\), and \(\text{TRANrep2}\) to define the state-transition table. To clarify, Transition (A,b,B) is shown in the state-transition diagram as a TranRep1 object, displaying an arc between nodes A and B with label b, and in the state-transition table as a TranRep2 object displaying the ToState name, B, in the table entry for state A and token b.
Execute(Action),
Return(ToState);

Figure 6. Transition Classes

Figure 8 shows the object interconnection model for SimpleFSA along with its state-transition diagram representation. The left side of the diagram shows the relationships between the FSA object, state objects and transition objects. The Scope of both the states and transitions is the SimpleFSA object, which has access to its states and transitions through its States and Trans fields. These interconnections graphically illustrate the structure for the functional model of an FSA. The right side of the diagram shows the relationships between the various reps. The MetaRep/SubRep relationships define the composition of the graphical presentation of the state-transition diagram for SimpleFSA. The links that cross the center of the diagram define the Reps/Item interrelationship between objects and their representations. A similar object interconnection model exists for the SimpleFSA and its state-transition table representation.

Executing an FSA

The execution of an FSA is defined in terms of the script for the FSA object. For example, if an FSA receives the message \texttt{RUN(FSA,input)}, then the \texttt{RUN} method (found in the script of the FSA object) is executed. Referring to Figure 3, the \texttt{RUN} routine defined in the FSA object finds the transition \texttt{tran} from the current state \texttt{curr-state} matching the trigger \texttt{input}. It then sends a \texttt{DEACTIVATE} message to the state object \texttt{curr-state}. Next, a \texttt{RUN} message is sent to the transition object \texttt{tran}, which sets the new current state. Finally an \texttt{ACTIVATE} message is sent to the new \texttt{curr-state}. The \texttt{ACTIVATE} and \texttt{DEACTIVATE} routines for state objects, described in Figure 5, send \texttt{HIGHLIGHT} and \texttt{UNHIGHLIGHT} messages to all the rep objects associated with them. The \texttt{RUN} routine for the transition object, described in Figure 6, simply executes the \texttt{action} associated with the transition and returns its \texttt{ToState}. For example, if SimpleFSA receives the message \texttt{RUN(SimpleFSA,b)}, and the current state is \texttt{A}, then transition \texttt{(A,b,B)} is found and its \texttt{action} is executed making \texttt{B} the current state. In the process, state \texttt{A} is \texttt{DEACTIVATED} and state \texttt{B} is \texttt{ACTIVATED}, causing the corresponding representations to unhighlight and highlight, respectively.
Manipulating an FSA

Modifications to a defined FSA can be demonstrated by adding a new transition to the SimpleFSA. By typing B in row c, column B of the state-transition table, a request for a new transition \((C, b, B)\) for SimpleFSA is made. To accomplish this, the FSArep2 rep sends a NewTransRequest message to its Item, the SimpleFSA object. SimpleFSA then determines the correctness of the request; i.e. a valid transition, instantiates a new transition object and makes the appropriate transition/scope interconnections. SimpleFSA then sends a NewTrans message to each of its Reps, FSArep1 and FSArep2. These reps now instantiate and position the corresponding transition subreps appearing in them as well as establish the Item/Reps links between the subreps and the new transition object. The new transition \((C, b, B)\) now appears in both views of SimpleFSA.

Current Work

An initial prototype of the representation framework has been developed on an Apple Macintosh II, using an object-oriented extension of C. In order to test the framework, the FSA example was implemented. A functional model describing an FSA was created, and two representation models, diagram and table, were
defined. The prototype FOX (for Fsa Object eXample) helped identify some key issues that currently are being addressed. We discovered that multiple displays of the same representation would add to the richness of the model. For example, under the current framework there is no provision for consistent displays of representations of the same type. If two diagram representations for SimpleFSA were created, the moving of a node in one diagram is not reflected in the other. FOX demonstrated the power in separating the functional model from the set of representations of an object. The prototype demonstrated the feasibility of providing multiple, concurrent, graphical views in an efficient and practical manner. We are currently reimplementing the framework on an Apollo 3500 Workstation using C++. We are testing the framework on other functional models including relational databases and a development environment.

**Conclusion**

The goal of the representation framework is to provide a mechanism for users to specify a functional model, its visual representations, and its interrelationships with other objects. The above example demonstrates how the underlying objects, their interrelationships and evaluation are used in the framework. The prototype FOX has encouraged us to look at extensions to the framework in anticipation of supporting future graphical environments.
Acknowledgements

The initial prototype for the framework was developed at Digital's Artificial Intelligence Technology Center in the Intelligent Systems Technologies Group. We are continuing to develop the framework with the support of Nixdorf Computer Engineering Corporation. We would like to thank our former colleagues for their comments and reviews of our work and acknowledge our current colleagues for their continuing support and encouragement.

References


