GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views

Michael F. Kleyn and Paul C. Gingrich
Schlumberger-Doll Research
Old Quarry Road
Ridgefield, CT 06877-4108
USA

Abstract:

Object-Oriented programming is a powerful means of developing large complex systems. In this paper we address the need to understand the behavior of objects in order to facilitate code sharing and reusability. We describe *GraphTrace*, a tool we have developed that has allowed us to experiment with new ways of visualizing the dynamic behavior of object-oriented programs. Based on our experience with the GraphTrace tool we suggest that being able to present many different views of an object-oriented system and then animating these views concurrently represents a powerful means for understanding such systems.

1 Introduction

This paper describes GraphTrace, a tool we have developed to assist in understanding object-oriented programs. GraphTrace allows a user to create displays revealing different aspects of the structure of an object-oriented program, and then to animate these displays in order to visually understand how the program works. In this paper we show how different displays (or *views*) can be created and animated with the tool.

We use mainly (though not exclusively) graph diagrams to display the structure of programs. Graphs consist of nodes and links. In general the nodes represent components of the system, such as objects, methods and instance variables, and the links represent some relationship or set of relationships that exist among the components, such as inheritance and delegation.

Animation is achieved by highlighting and dynamically annotating the nodes and links to show the activity of the program. We have found that animation, even in this simple form, is an effective technique for helping a developer understand how a system works.

The paper is divided into four parts. In Section 2 we motivate the need for tools for understanding object-oriented systems by recognizing that the programming methodology associated with the reuse and extension of existing objects pre-supposes the designer has some understanding of the behavior of the objects. We show how GraphTrace builds on previous work to provide a tool for understanding both the detailed workings and the large scale operation of object-oriented systems. Section 3 provides a description of how our system is implemented and how it is used. It includes a simple illustrative example of GraphTrace in operation. In Section 4 we explore a detailed case study in which we show how multiple concurrent GraphTrace views can explain the workings of an example system. Finally, Section 5 compares GraphTrace to previous work in the area of program animation and diagramming message sending in object-oriented programs.

2 Background and Goals

We briefly review the important characteristics of all object-oriented languages. Objects have both *methods* and *local state*. Methods are the operations that define an object's behavior and an object's local state is the current set of data values maintained by the object. Object-oriented programs execute through message passing among objects. The methods in an object execute in response to a message and may then invoke further messaging. An object-oriented language also includes the concepts of *class* and *inheritance* where objects of the same class have the same methods and classes may inherit operations from superclasses [Weg87]. Connections between objects represent many different kinds of relationships, including

the most commonly used *is-a* and *part-of*.

We address the problem of understanding and explaining the behavior of large-scale object-oriented systems. An application system written in an object-oriented programming language contains many complex objects linked in a variety of relationships. The intent in using an object-oriented programming language is to model a *richly structured application domain*. It is for this reason that object-oriented languages are increasingly being used as the medium in which *knowledge-based* systems are designed. A typical application program can include hundreds of objects and hundreds of methods defining the operations for those objects.

Understanding and explanation are general issues that arise in the concerns of program usage, education, technology transfer and object reusability. An explanation facility for a system is an integral part in ensuring that the system is understood and used correctly [Bro78]. Swartout's work has emphasized that program explanation is one of the key features of expert and knowledge-based systems [Swa83]. Consider also the problems involved in technology transfer, when it is necessary for programmers to expand or enhance an object-oriented system they did not design.

Object reusability is the characteristic that a well defined object designed for one application can be used in a similar application. One can also reuse objects in the same system by creating subclasses that will inherit the operations of superclass objects (code sharing). Specializing an object can also involve the overriding of default states and methods. New methods are then written so that the specialized object behaves in a slightly different manner. The reusability scenario, however, often involves some understanding of the specific behavior of existing objects. Documentation is one answer to this problem but documentation rarely keeps pace as an object's methods are altered over time. Furthermore, static descriptions are often inadequate for the task of conveying an understanding of inherently dynamic processes.

Simply executing a computation is not enough to explain a system's behavior. One must understand both the structure and the function of the objects involved. One must understand the taxonomic structure of the class objects, the inheritance mechanisms used, the individual behaviors of objects, and the dynamic behavior of the system as a whole.

The problem is somewhat analogous to that of viewing a sports event such as as tennis or football game. Many different camera angles are required to provide an understanding of the action taking place. Each camera reveals particular aspects of the action that could not be conveyed by one camera alone. Furthermore it is the combination of views of the same action that provides an even more complete understanding.

In a similar manner, one must examine an object-oriented system from many different views. We define a *view* to be a user's interface into an object or set of objects. A taxonomy graph is a view of the class-subclass structure in a knowledge base. Likewise, the *part-of* relationship for the same objects may be viewed in a part-whole graph. A window into the contents of a single object is also a view. The important point is that the structures and the links that define relationships are already in place - they are part of the original system design.

Because systems have become increasingly large and complex, developers have turned to software tools to record and report on the internal state of a program in operation. Developers are increasingly turning to the use of high-resolution bit-mapped displays and interactive graphics as vehicles for their software tools. This has lead to experimentation with program and algorithm animation as a basis for program explanation and comprehension. Animation for programs draws from work in the field of graphical animation techniques [Rae85,BS84] and from work in program analysis [MM85]. Environments to support programming in specific languages sometimes include animation; see [Bar86] for an example in dataflow program development. The work described in this paper builds on this body of work in program animation.

By collecting and concurrently animating the differing views of an object-oriented system, a graphic visualization of the dynamic behavior of the entire system can be provided. The GraphTrace facility enables the user to easily produce animated views of his or her program; these views resemble a *motion-picture* of the entire system's dynamic behavior.

GraphTrace was developed within the powerful object-oriented programming environment that we use. The environment consists of two major parts, *Strobe* and *Impulse-86*. Strobe, is an object-oriented language implemented as an augmentation to Common Lisp. We have built a large number of systems in Strobe and one of the reasons for developing GraphTrace is to be able to show colleagues how these Strobe programs work. Strobe is a language that provides many advanced features such as message passing, multiple inheritance, and method combination. In Strobe an object is composed of *slots* where a slot may represent either a method or a state variable. More details of the architecture of the Strobe language can be found in [SC86,Smi83,SS83]. Impulse is a substrate for build-

ing user interfaces. It is itself implemented as a collection of Strobe objects so that building user interfaces consists of modifying, specializing and extending the objects that make up Impulse. Impulse was used to implement the user interface of GraphTrace. A large number of interfaces have been built in Impulse including extensions for graphical interfaces, interactive data editing and forms interfaces. The workings of these interfaces, also termed *specialized editors*, will be used as examples of object programs that can be analyzed with GraphTrace. More details on the implementation of Impulse and how it is used to create user interfaces are found in [SDB86,SBY87,SK87,RGSD87].

3 The GraphTrace Tool

This section describes how GraphTrace is implemented and how it is used. Figure 1 shows a typical GraphTrace display. On the left is an application program which is running. The application program is the *Ida* system, an interactive system for displaying and viewing 2-D data plots [You87]. This snapshot was taken just after the user changed some of the values in the data and invoked a "refetch" command to redraw the display. The display has just been cleared in preparation for redrawing the data plot.

The displays on the top and on the right are two GraphTrace views of the application program. The view on the right is the *method invocation* graph, a view of the sequence of methods that are invoked as a program runs. The method invocation graph is one of the most used GraphTrace displays and is described in more detail in Section 3.2. The view on the top of the figure is a taxonomy graph of some of the objects that implement Ida. The highlighting of links and nodes in the graphs show which parts of the system are involved in computation at the current time. In this particular case the node `Clear` representing the method used to clear the display is highlighted to show that it is currently being invoked.

3.1 Using GraphTrace

GraphTrace is operated via the control panel menu shown in Figure 2. There are two phases to using GraphTrace - *recording* and *animation*. The user first sets GraphTrace into recording mode by selecting the recording mode item in the `Record/Animate` menu. An application program (or operation within an application) that the user wishes to investigate is then run and GraphTrace records information about the messaging activity that occurs as the program runs. When completed, GraphTrace is set to animate mode and a view of the recording is displayed on the screen. The application program is then re-run and GraphTrace causes the recording and other active views to be animated.

The speed at which the animation is re-run is controlled with the right-side menu. The animation can be slowed down or speeded up as the program is running. This allows, for example, the user to quickly run the animation at `FAST` speed and then to change to `MEDIUM` or `SLOW` as the program approaches the point in the computation that is of interest. Since the animation by GraphTrace consists only of simple graphical operations, the speed of running an application with GraphTrace is very close to that of the program running by itself (this is true for both recording and animation). `SINGLE STEP` can be used to cause the program to halt before each message is sent.

The GraphTrace graph displays include a pop-up menu of commands to selectively prune the graph so that the user can zoom in on activity in particular parts of the graph. GraphTrace graphs can be stored on file so that the displays can be regenerated and reanimated at a later time.

3.2 GraphTrace Implementation

GraphTrace works by temporarily modifying the low-level lisp function that implements message passing in Strobe. Two modifications are alternately applied. In the recording mode, the function is made to record information about the context of message invocation as a program is run. This information includes the originator of the message, the recipient object of the message, the name of the method that was invoked, and the values of arguments passed in the message. This information is stored and then used to create graph displays. In animation mode, the lisp function is modified to cause animation of the displays as the program is re-run.

This kernel component of GraphTrace does not *require* the animation to occur in a graph display. Indeed, any editor built with Impulse can be interfaced to GraphTrace and animated if it has methods to highlight components of its display. Most editors do have such methods which are used for interaction purposes. In this paper, however, we concentrate on graphs since we have found animated graph displays to be a compact and expressive representation for understanding program behavior. The GraphTrace graph displays fall into two general categories – *structural* and *behavioral*:

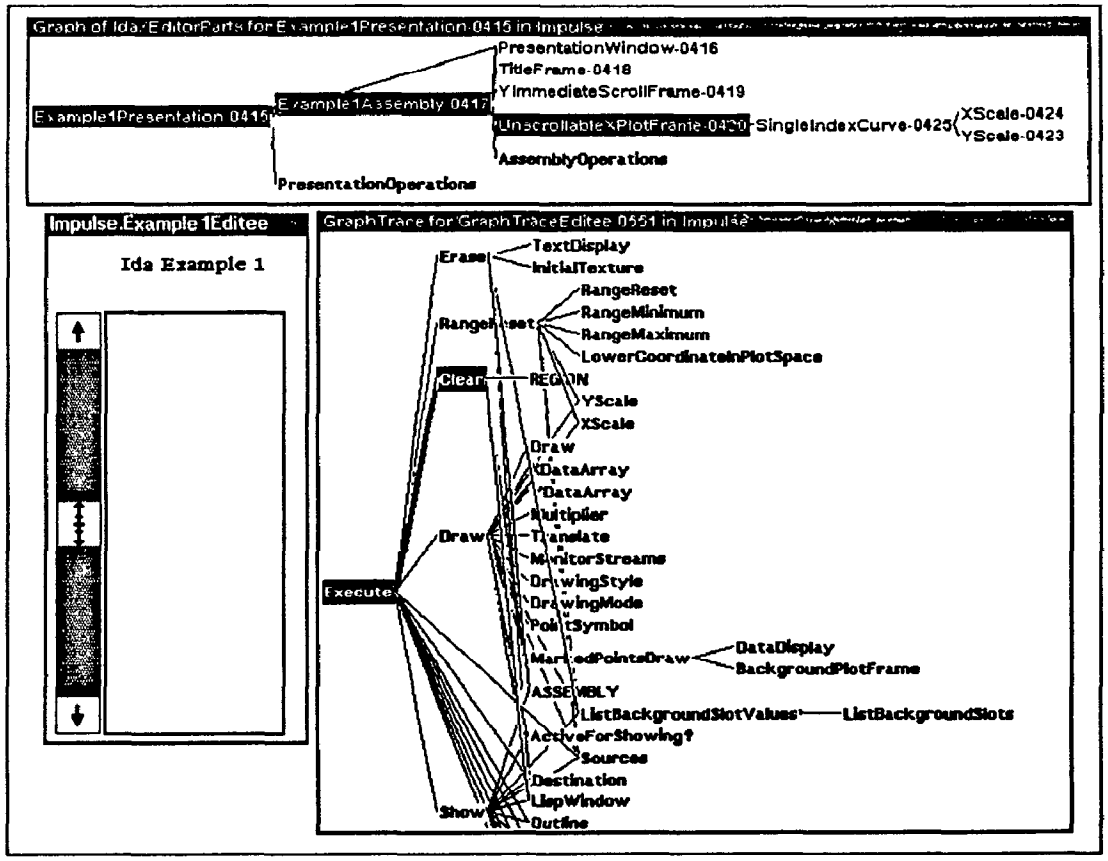

Figure 1: An Analysis Session with two GraphTrace views

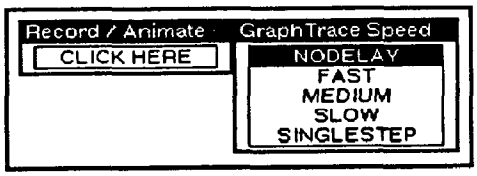

Figure 2: GraphTrace Control Panel Menu

3.2.1 Structural Graph Displays

A structural display is a view of static structure in a program. Graph displays showing the static structures that exist in a Strobe program are available as part of Impulse. With GraphTrace, these displays are augmented with animation to show how the running of a program relates to the particular structure being displayed.

For example, the GraphTrace taxonomic displays show how activity is partitioned across the class hierarchy. Taxonomic displays include both *Progeny* graphs which show the standard taxonomic hierarchy and *Ancestry* graphs which "look up" the inheritance hierarchy towards the root object. In an ancestry graph, it is possible to show how the inheritance structure is being exploited by highlighting the class object in which a method is defined when the method is invoked.

Another type of structural display is the part-whole graph. In Strobe a built-in *part-of* construct is not available, but program designers can easily create their own application dependent *part-of* links. To then display a *parts* relationship the designer creates his or her own Impulse structure graph by defining a *generator* function. The generator function specifies how to generate a part-whole relationship graph from an object with the appropriate links. GraphTrace can also animate these graphs.

3.2.2 Behavioral Graph Displays

Behavioral graphs show the dynamic relationships that occur as the program is run and messages are exchanged. Behavioral graph displays can be thought of as displays of "run-time structure" since they show the temporal (and temporary) connections that occur between components of the program. A *method invocation* graph is one kind of behavioral view. It is a condensed view of the sequence of messages that are invoked as a program runs. In this display, the child nodes of a node show the methods that are invoked by another method. The graph as a whole shows the complete nesting of message invocations.

An invocation trace is a program debugging display which is similar to, but more concise and evocative than a standard program trace. In a standard "terminal output" program trace some facility (or the insertion of "print" statements in the code) allows the programmer to cause the program to have the side effect of printing some information into a text window. A typical printout trace might look like this:

```
⇒ Entering Method1
      ⇒ Entering Method2
            ⇒ Entering Method3
            ⇒ Exiting Method3
      ⇒ Exiting Method2
      ⇒ Entering Method4
      ⇒ Exiting Method4
⇒ Exiting Method1
```

Such a terminal output trace is useful for showing details of a program's activity, but one can get lost in several pages (or screens) of indented method invocations. Invocation tracing is simply the idea of using a tree display to represent this same information. Thus, an invocation trace would appear in a "method invocation graph" as shown in the Figure 3.

The tree displays the nesting structure of messages. While a program is running, the method names in the graph are "flashed" by highlighting a name and/or link on entry and then unhighlighting it on exit. This representation is more compact than a printout. The tree shows a complete history of the invocation stack. Each step in a depth-first traversal of the tree corresponds to the successive states of the stack as the program runs. The path of highlighted links and names from the root indicates the current stack state of the program. We can deduce from Figure 3.2.2 that the stack goes through the following six states:

```
                Method3
          Method2 Method2 Method2 Method4
Method1 Method1 Method1 Method1 Method1 Method1
1       2       3       4       5       6
```

Many programming environments have a *break* facility [Int85]. In a typical break package it is possible to stop the computation to view the current state of the stack, and see what the current nesting of function invocations is, but it is not possible to see the past state of the stack when other functions were on it. An invocation trace graph does allow the user to see this past history, branching off from the current "stack state path." It gives a better overall view of where the program has been and where it is going. In the example of Figure 1, the current stack state path is **Execute, Clear**.

Note that we display a method invocation as a graph, not a tree, so that repeated invocations of the same method are represented by a single node, and not several nodes of the same name for each invocation. The repeated highlighting of the single node during animation shows repeated invocations. We do, however, show two nodes of the same name to indicate recursion; a node that has a child node of the same name indicates that the method invokes itself.

Programming environments often also provide cross-referencing or code analysis facilities that can display

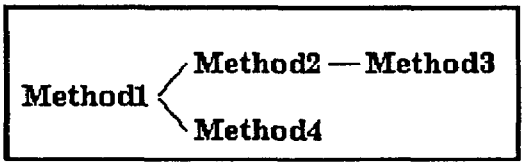

Figure 3: Invocation Graph

the graph of method invocation nesting by deducing it from a static analysis of the code (such as the Interlisp Masterscope Browser tool [Int85]). In the case of an object-oriented message passing system such as Strobe, binding of methods to Lisp functions occurs at run-time, and so an invocation trace is the only way of acquiring a view of the functions as they are actually invoked. The invocation graph can be thought of as the object-oriented programming equivalent of a calling or function nesting graph in functional programming languages.

Another example of a behavioral graph display is the *object invocation* graph. This is similar to the method invocation graph, but shows only the names of the objects involved without indicating the particular method that was involved. It gives a large-scale view of the overall message traffic between objects. Another behavioral graph is the *object-slot invocation* graph in which each node is and object, slot pair identifying a specific method in a specific object. This provides a detailed view that will highlight different nodes when the same method is invoked in different objects.

The behavioral graph displays are generated at run time which is why a two-phase record and animate protocol is needed for showing these displays – the first to create the graph and the second to animate it. Structural graphs can often be derived from the static structure of the object program and so they can be generated without a preliminary record phase.

One advantage of using such a simple medium for showing animation is that it is easy to create new displays and animate them. Particular object-oriented programs may have unique structures in addition to the *is-a* hierarchy and method invocation nesting. The next section describes an example analysis session with GraphTrace and includes a variety of displays.

4 Understanding a System Using GraphTrace

In this section we present an example of how GraphTrace is used in the understanding of a sophisticated object-oriented system. The application we examine is the user interface substrate Impulse. As previously mentioned, Impulse is a general and extensible substrate upon which a large variety of interactive user interfaces have been developed. In particular, editors have been developed in Impulse that are used for editing the internal structures of Strobe objects. The example we present is the creation of a *fast object editor* used for editing a single Strobe object. The incremental creation or *instantiation* of a fast object editor inspecting the `LoggingDemo` object is presented in Figure 4. The first five lines of the completed editor show the values of properties that are common to all objects and the remaining lines show the local methods and state variables. The slot `LOG` is a method (indicated by the `[LISP]` annotation immediately after the name of the slot) and the name of the method handler is `IMPULSE/LOGGINGDEMO/LOGWELL`. `RIG` and `TRUCK` are local state variables with bitmap values.

An editor in Impulse is built using the five basic classes `Editor`, `EditorWindow`, `PropertyDisplay`, `Menu`, and `Operations` as shown in Figure 5. These classes are connected in a uniform framework and each class has a well-defined set of operations and responsibilities [SDB86]. The `FastObjectEditor` class defines the *fast object editor*. It is a specialization of the `ObjectEditor` class which defines the *standard object editor*. The name derives from the fact that time is saved by not instantiating the command menus associated with the editor. These menus normally appear attached to the side of the window with the standard object editor but are only instantiated and popped-up upon user demand in this specialized editor.

The first step towards understanding the fast object editor is to locate the object in the Impulse taxonomy (Figure 5). This view of Impulse is not the entire hierarchy but is a subset, selected according to membership in the group `FastObjectEditor`. The objects that constitute the fast object editor are boxed. It can be seen that these objects are specializations of the object editor for all five of the major Impulse classes.

In Impulse one can inspect relationships other than *is-a* through the use of user-defined structure graphs.

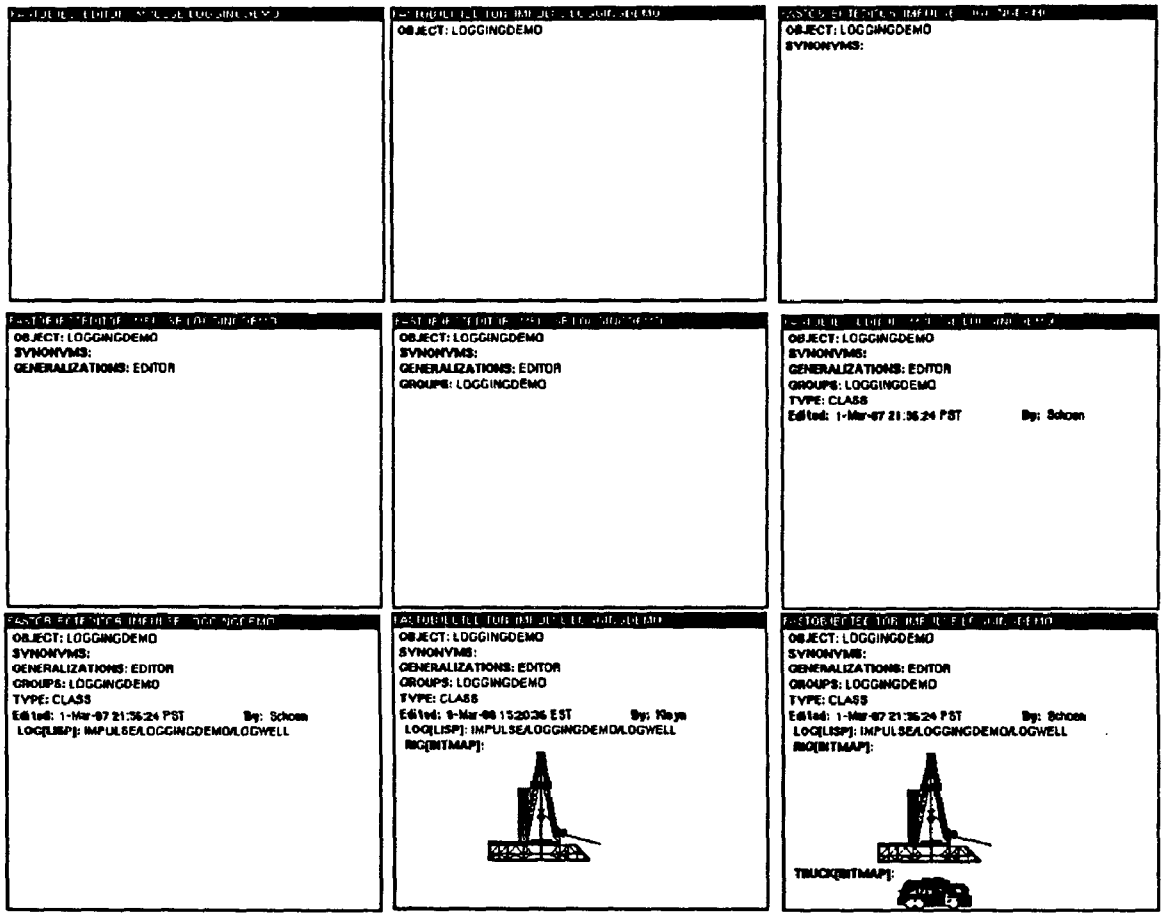

Figure 4: Progression of a fast object editor view instantiation

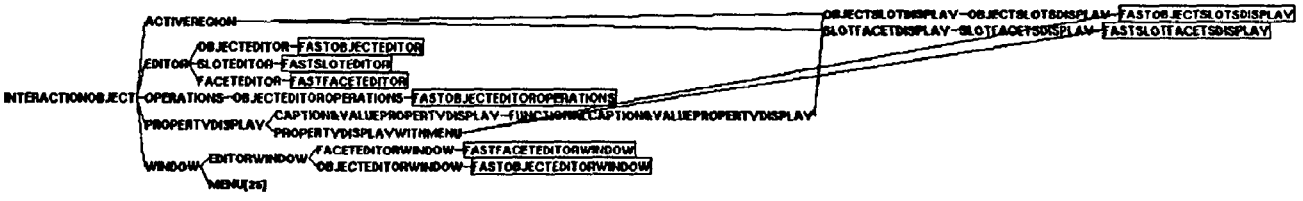

Figure 5: Subset of Impulse taxonomy

The associated structure graph for the fast object editor is called *EditorParts*. These links were created by the original designers indicating that the *part-of* relationship is an important aspect of this editor. Selecting the parts view yields a graph of the components of the fast object editor (Figure 6). It can be seen that the fast object editor is composed of a window (the `FastObjectEditorWindow`), several displays, a slot editor and an operations object. The `FastObjectSlotEditor` component is itself an editor with its own components. With the use of GraphTrace, we will show how the `FastObjectSlotsDisplay` is responsible for the actual printing of the slot names and values in the editor window when the editor is invoked.

At this point we have discovered where the `FastObjectEditor` exists in the Impulse taxonomy and the objects that constitute this editor. We can examine this object, on the left in Figure 7, and see that only the `Components` and `Operations` slots are locally defined. All of the methods defining the behavior of this object are inherited from its superclasses. We can examine some of these methods (on the right in Figure 7), but we can not be certain which of the methods are used during editor creation. Methods also exist for interactive updating and editor deletion. Visually inspecting an object, therefore, will not explain the dynamic behavior of that object during a given process. One can always examine the method handler code in detail but this would entail constructing the sequence of method invocations by hand. A dynamic trace of the instantiation of a fast object editor view is needed in order to determine which methods are used during the creation process and the sequence in which they are invoked.

Figure 8 is a GraphTrace method invocation graph displaying the names of the `FastObjectEditor` methods that are invoked during the instantiation of the fast object editor. Tracing and animating this graph during the instantiation of the editor reveals the underlying operations of this editor class. The graph indicates that the current stack state includes the `FastObjectEditor` methods `CODE`, `InstantiateComponents` and `EditorWindow`. We have only included the methods that are inherited by the `FastObjectEditor` but we can also produce a graph displaying all of the methods that are used during the instantiation process (Figure 9). Figure 9 displays a different sequence of methods because GraphTrace has recorded methods from all of the fast object editor component objects.

As explained in the previous section, we can produce other types of invocation graphs each with its own particular view. For example, when the `FastObjectEditorWindow` creates a lisp window, the `CreateLispWindow` handler sends several messages to collect local state. We can examine the `CreateLispWindow` handler more closely and let GraphTrace display the actual values being sent to and returned by the subordinate methods (Figure 10). This invocation graph shows that messages are being sent to the `InitialWidth` and `InitialHeight` slots to retrieve the initial dimensions of the editor window.

Strobe uses the Flavors [SB86] idea of *mixins* for methods combination, that is, one can successively execute a series of methods that are linked via inheritance. In Strobe the *top-down additive inheritance* mechanism can be used to execute one method and then proceed down the class taxonomy executing methods of the same type. For example, a top-down additive inheritance message sent to the `Initialize` method of a `FastObjectEditorWindow` instance will invoke all of the `Initialize` methods from that editor window's highest superclass to the most immediate superclass. This process of method execution can be viewed in an animated GraphTrace ancestry graph (Figure 11).

We have examined various aspects of the `FastObjectEditor` and the creation of a fast object editor view, from a global taxonomic perspective to the more detailed level of methods and the values they return. By animating all of these views concurrently as an editor view is created we can see a graphic visualization of the entire editor instantiation process. Figures 12 and 13 are screen snapshots of GraphTrace animating different views as a fast object editor is created. These figures correspond to the first and second last pictures in the sequence of Figure 4.

On the bottom of the screen is the taxonomy graph. On the left is the method invocation graph. To the right of the invocation graph are, from top to bottom, the parts view, the `FastObjectSlotsDisplay` object and the fast object editor that is being examined as it is being created. We can see that the editor is at the point where the window has just been created and is displayed empty on the screen. Highlighted nodes indicate that a method or object is taking part in a message invocation.

From the structure graphs we can tell that slot display is accomplished by the `FastObjectSlotsDisplay` object. As the `RIG` slot is displayed, (Figure 13), the `FastObjectSlotsDisplay` object is highlighted in both the taxonomy and parts graphs. Concurrently, the `Display` method is being highlighted in both the invocation graph and the animated `FastObjectSlotsDisplay`. From the method invocation

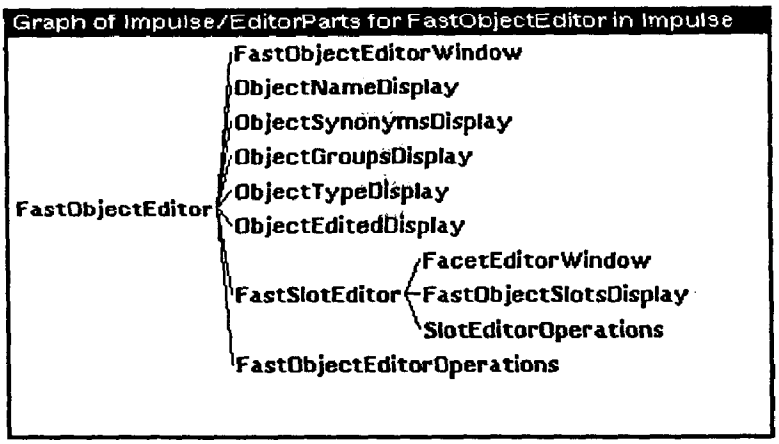

Figure 6: `FastObjectEditor` Components

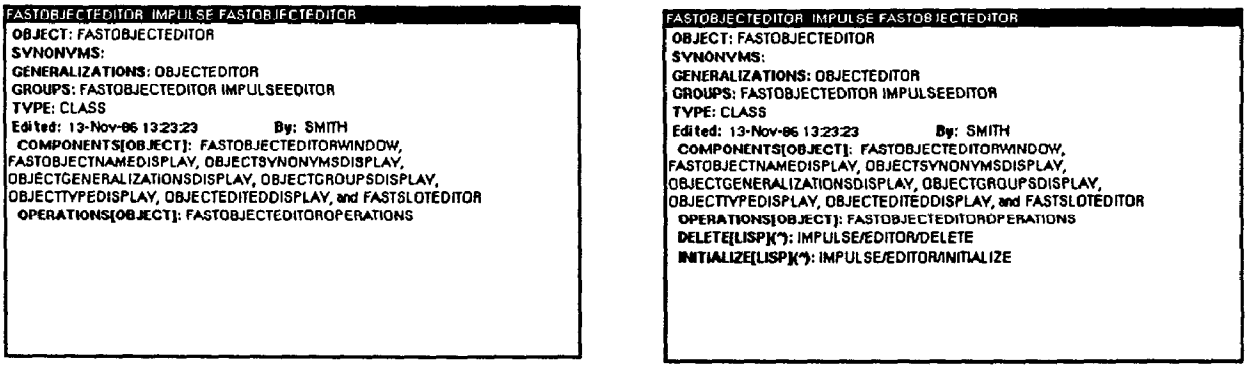

Figure 7: The `FastObjectEditor` class (left) with inherited methods (right)

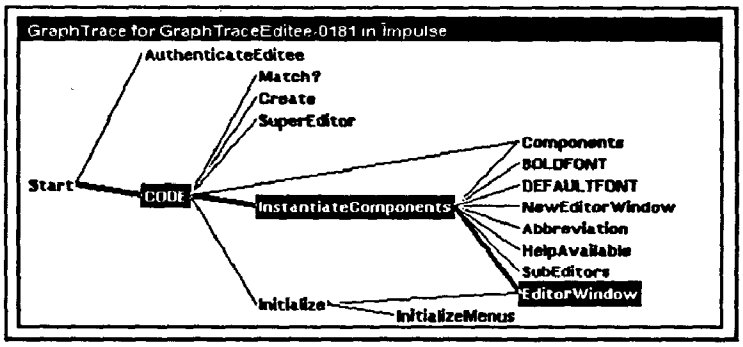

Figure 8: Fast object editor method invocations – `FastObjectEditor` methods

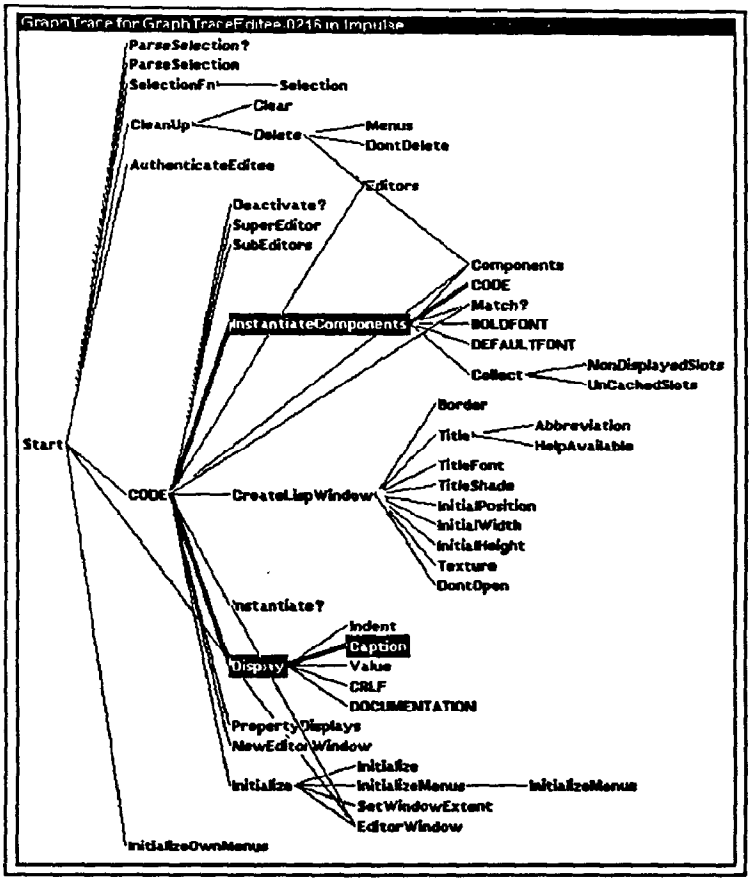

Figure 9: Fast object editor method invocations – all methods

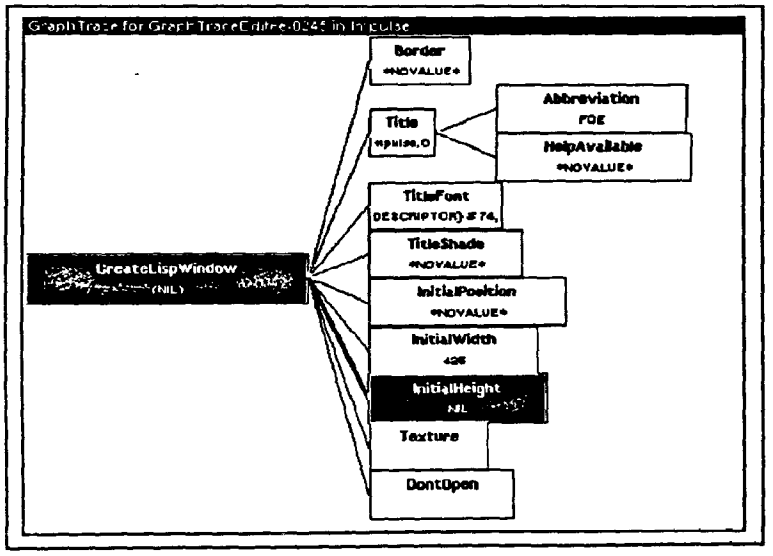

Figure 10: `CreateLispWindow` method invocations with arguments

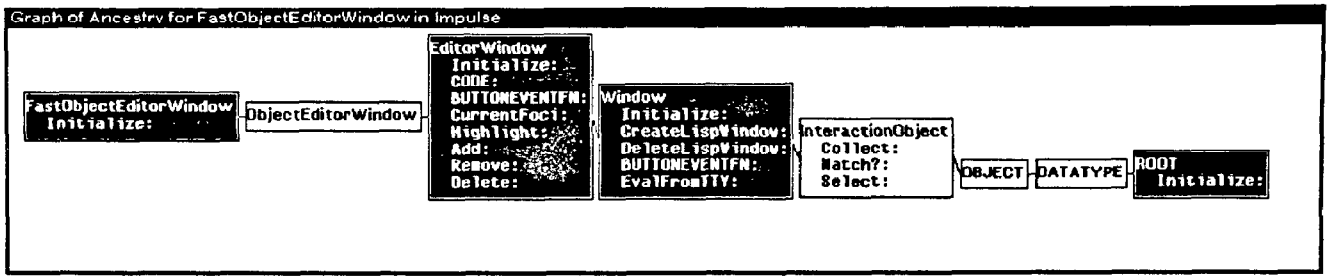

Figure 11: Combination of *Initialize* methods

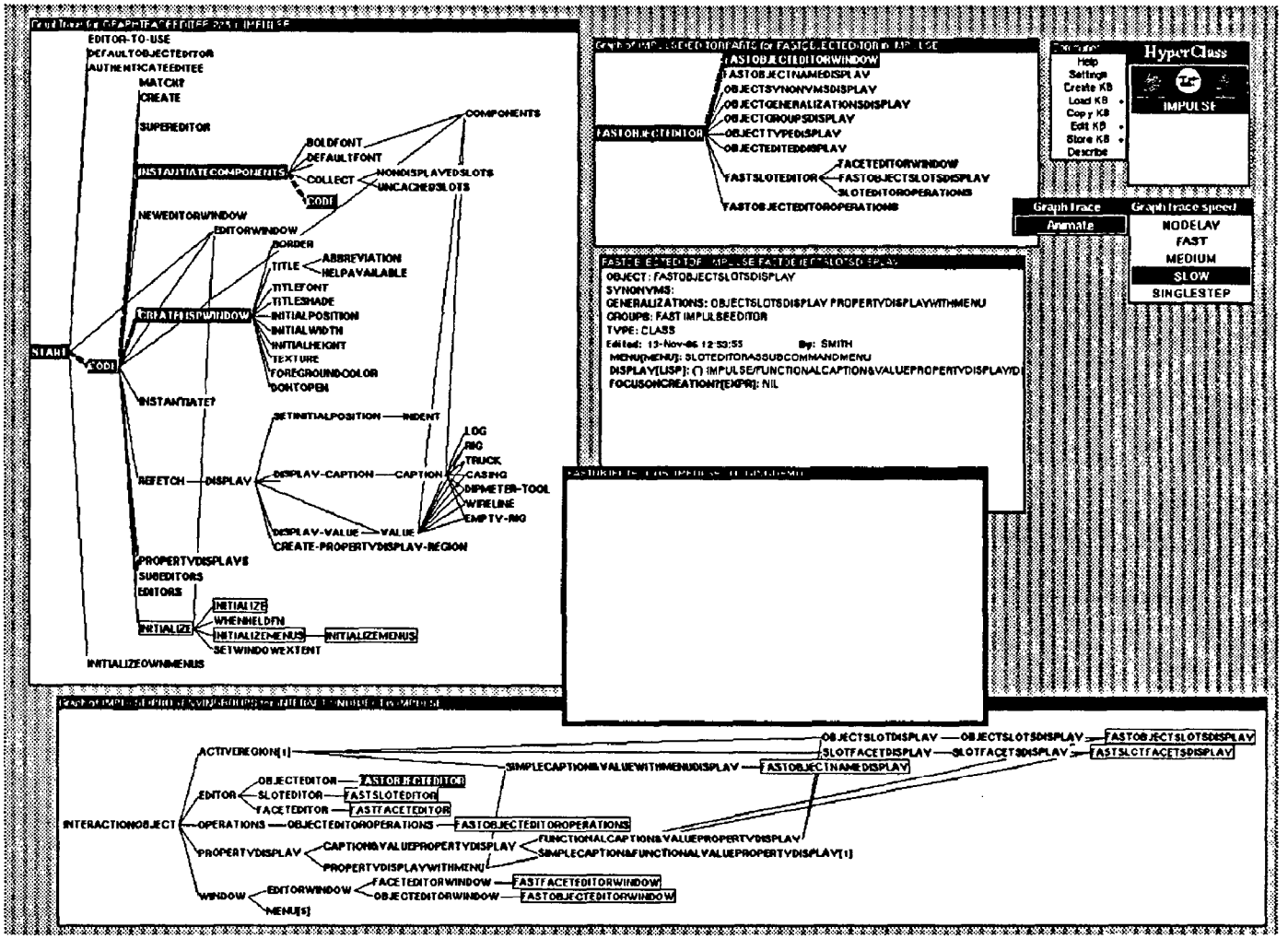

Figure 12: GraphTrace screen snapshot 1 - Displaying LOG slot

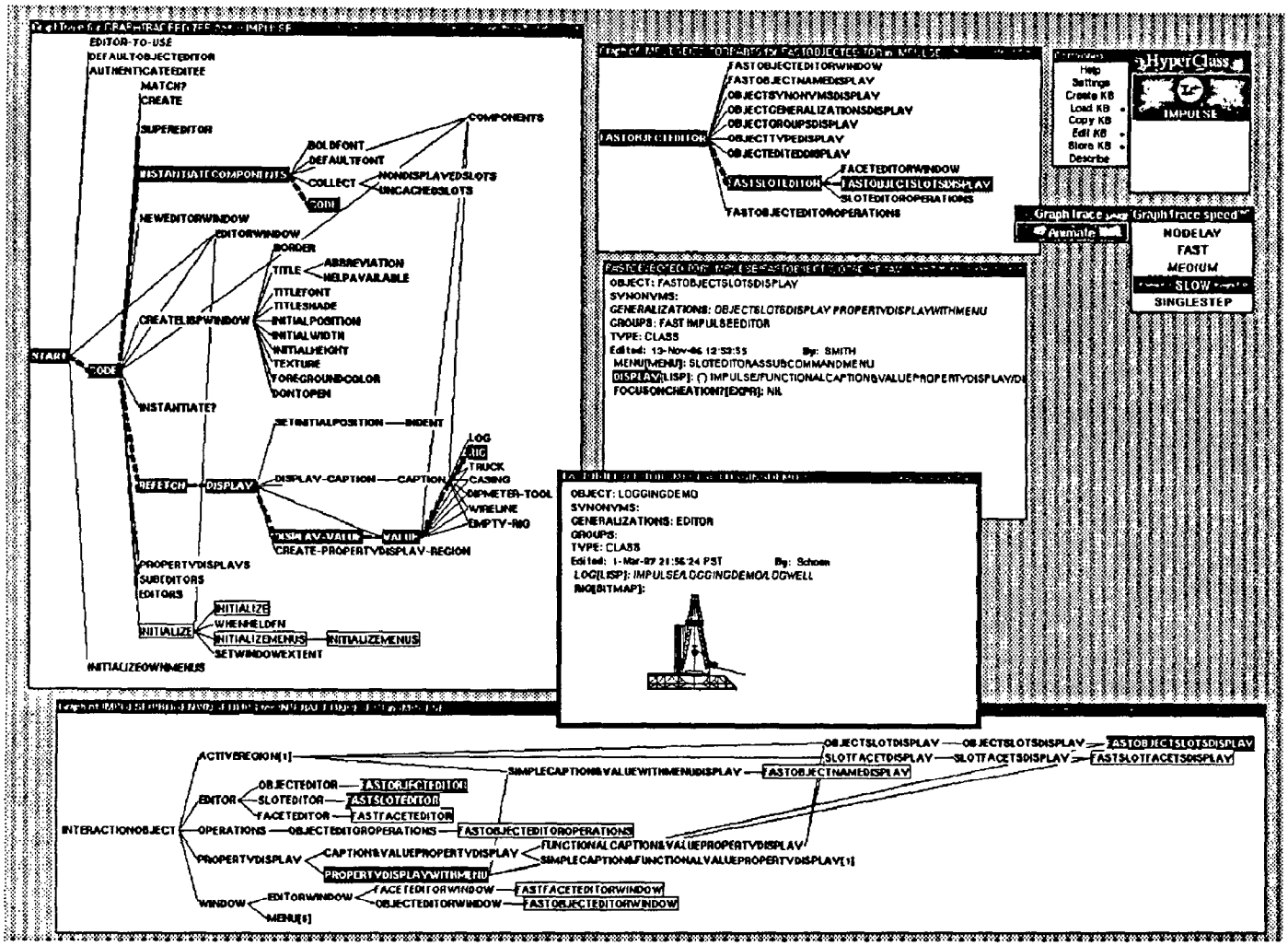

Figure 13: GraphTrace screen snapshot 2 - Displaying RIG slot

graph we can see that the invocation stack includes the `Code`, `Refetch`, `Display` and `RIG` methods. The `Display` handler is now messaging the `RIG` slot in the `LoggingDemo` object, presumably to retrieve its value, which is then painted onto the screen.

The name "FastObjectSlotsDisplay" is certainly an indication of the object's function but cannot explain the object's entire behavior. Note that a user would not know to display this object on the screen until he ran the trace once and saw this class object being highlighted during slot display. After the `RIG` slot is displayed, the `Display` handler will proceed to paint the next slot, `TRUCK`.

Animating these views yields an enlightening motion picture of the system in operation. The important point about these views is that they expose the natural views of the system's structure; they are the objects and relationships built by the original designers. The method invocation view is complementary to the taxonomy and parts view. Each view presents a different perspective on system behavior and each perspective yields different information. By using Impulse as our example object oriented system we have tried to give the reader a flavor of how GraphTrace can be used to browse and gain understanding of an unfamiliar system. Certainly a user would have to examine other aspects of Impulse in order to gain a complete understanding.

Using the GraphTrace facility is analogous to using a standard trace package. The user has to first identify the particular views of interest. We have found that as designers add different structures and relations to a knowledge base they include the generators necessary for instantiating and displaying them. With GraphTrace, structural views are just instantiated and animated while behavioral views are recorded and then animated. Without having to resort to complex graphics programming, we can understand and explore a sophisticated object-oriented system.

5 Related Work

The extensive body of work in program animation is surveyed by Raeder in [Rae85]. The work that is most closely related to ours is that of London and Duisberg [LD85] and Cunningham and Beck [CB86]. London and Duisberg present an algorithm animation kit using the Smalltalk programming environment [Gol84]. Their method of program animation involves the insertion of graphics code into existing code. *AnimationView* and *AnimationController* objects are created to handle the animation of individual algorithms. The view or interface is constructed by the animator and may be useful for the animation of similar algorithms.

In GraphTrace, we create the animation by inserting hooks into the low-level message receiver. By simply monitoring the message traffic in the application system there is no additional programming that needs to be done by the user. The views that we animate are the natural interfaces to the structures already designed into a system. The animations done by London and Duisberg are more sophisticated, but they also require extensive graphics programming. Also, London and Duisberg are concerned with the animation of individual algorithms while we are concerned with both the detailed workings and the large scale behavior of a system.

Cunningham and Beck present a method for diagramming the message activity that takes place during an object-oriented computation. They use directed arcs to represent messages between objects of different classes. Their notation, however, does not explicitly represent the sequence of a computation - although they do mention that they have looked at this. The GraphTrace invocation graph is a direct representation of the sequence of message calls. Cunningham and Beck are primarily concerned with the visualization of inheritance between classes. We feel that by animating different views of an object-oriented system at the same time the user gains a greater understanding of the relationship between the entire system's structure and function. With GraphTrace, the compactness of the invocation graphs enables a user to examine the behavior of a relatively large system on a single screen.

6 Further Work

We are currently investigating two slightly opposing extensions to GraphTrace. The extensions are along the lines of *debugging* and *visualization*. We are experimenting with GraphTrace as a debugger, so that when a program has been stopped at a particular point in the computation, the user can select a node in the invocation graph and see typical break information. This extension aims to make GraphTrace a central component in a unified debugging system in which tracing, breaking and program development all occur through a single interface. GraphTrace could also function as a performance analysis package. Currently we only keep track of the method invocations but we could also generate statistics such as the number of method invocations, the number of inheritance searches and the average length of an inheritance search.

The other extension emphasizes exploiting the visualization aspect of GraphTrace. We are investigating different ways to display the animation. We are currently working on a graph that combines the taxonomy and method invocation views. When an object receives a message we would like to be able to indicate the superclass where the message receiver actually resides. This idea is similar to Cunningham and Beck's work.

We use simple node names that represent object names, method names or method handlers. Using icons to represent the message traffic and convey the flavor of the domain in a domain-specific animation would move GraphTrace towards the work on algorithm animation. We feel that this has great value in providing "friendly" displays that would be useful for explanation and tutorial purposes. For system development and debugging we prefer an animation interface which is simple.

As can be seen by the example in Figure 12 we present graphs as the preferred display. There is no implementational reason to do so; any Impulse editor can be animated. By adding simple highlight methods to any specialized interface, an application programmer can make use of the GraphTrace animation kernel to expose the behavior of his or her program.

7 Summary and Conclusion

We have presented a tool for understanding the dynamic behavior of object oriented programs. The tool works with both structural and behavioral views of a system. Structural views are simply instantiated and animated. Behavioral views are generated by recording the message activity that occurs as a program is running and then animating those views. Concurrent animation provides the user with several perspectives on a system's behavior. We have already used GraphTrace to analyze some of our existing applications and have improved and repaired programs as a result of the analysis. In one test case we found that a particular handler was being called by mistake and should not have been involved in the computation at all.

The record/animate mode of operation is a choice between alternative approaches to program animation. We have chosen the technique of modifying the low-level function in our object-oriented language that implements message passing. This eliminates the work of having to modify the original application code. The grain of animation, therefore, is at the message level, which is appropriate for an object-oriented system.

8 Acknowledgements

We are grateful for ideas and suggestions we have had from discussions with Lee Metrick, Indranil Chakravarty, Paul Barth, Ruven Brooks and Patricia Carando.

References

[Bar86] P. S. Barth. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics Special Issue on User Interface Software*, 5(2):142–172, April 1986.

[Bro78] R. Brooks. Towards a theory of the ccomprehension of computer programs. *Int. Journal of Man-machine Studies*, 18:543–554, 1978.

[BS84] M. H. Brown and R. Sedgewick. A System for Algorithm Animation. *Computer Graphics*, 18(3):177–186, July 1984.

[CB86] Ward Cunningham and Kent Beck. A Diagram for Object-Oriented Programs. In *Proceedings of the First ACM Conference on Object Oriented Systems, Languages, and Applications*, pages 361–367, September 1986.

[Gol84] A. Goldberg. *SMALLTALK-80: The Interactive Programming Environment.* Addison-Wesley, Reading, MA., 1984.

[Int85] *Interlisp-D Reference Manual.* Xerox Artificial Intelligence Systems, Pasadena, CA, October 1985.

[LD85] Ralph L. London and Robert A. Duisberg. Animating Programs Using Smalltalk. *IEEE Computer*, 61–71, August 1985.

[MM85] B. Melamed and R. J. T. Morris. Visual simulation: the performance analysis workstation. *Computer*, 18(8):87–94, August 1985.

[Rae85] Georg Raeder. A Survey of Current Graphical Programming Techniques. *IEEE Computer*, 11–25, August 1985.

[RGSD87] M. F. Kleyn R. G. Smith and R. Dinitz. *Impulse Cookbook.* Research Note SYS-87-42, Schlumberger-Doll Research, October 1987.

[SB86] M. J. Stefik and D. G. Bobrow. Object-Oriented Programming: Themes and Variations. *AI Magazine*, 6(4):40–62, 1986.

[SBY87] R. G. Smith, P. S. Barth, and R. L. Young. A Substrate for Object-Oriented Interface Design. In B. Shriver and P. Wegner, editors, *Research Directions In*

Object-Oriented Programming, pages 253–315, MIT Press, Cambridge, MA, 1987.

[SC86] R. G. Smith and P. J. Carando. *Structured Object Programming In Strobe.* Research Note SYS-86-26, Schlumberger-Doll Research, October 1986.

[SDB86] R. G. Smith, R. Dinitz, and P. Barth. Impulse-86: A Substrate for Object-Oriented Interface Design. In *Proceedings of the First ACM Conference on Object Oriented Systems, Languages, and Applications*, pages 167–176, September 1986.

[SK87] R. G. Smith and M. F. Kleyn. *Impulse User's Guide.* Research Note SYS-87-39, Schlumberger-Doll Research, October 1987.

[Smi83] R. G. Smith. Strobe: Support for Structured Object Knowledge Representation. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 855–858, August 1983.

[SS83] E. Schoen and R. G. Smith. Impulse: A Display-Oriented Editor for Strobe. In *Proceedings of the National Conference on Artificial Intelligence*, pages 356–358, August 1983.

[Swa83] W. R. Swartout. XPLAIN: A System for Creating and Explaining Expert Consulting Programs. *Artificial Intelligence*, 21:285–325, 1983.

[Weg87] Peter Wegner. Dimensions of an Object-Based Langauge Design. In *Proceedings of the Second ACM Conference on Object Oriented Systems, Languages, and Applications*, pages 168–182, October 1987.

[You87] R. L. Young. An Object-Oriented Framework for Interactive Data Graphics. In *Proceedings of the Second ACM Conference on Object Oriented Systems, Languages, and Applications*, pages 78–90, October 1987.