Visualizing Interactions in Program Executions

Dean F. Jerding
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
dfj@cc.gatech.edu

John T. Stasko
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
stasko@cc.gatech.edu

Thomas Ball
Software Production Research
Bell Laboratories
Naperville, IL 60566
tball@research.bell-labs.com

ABSTRACT

Implementing, validating, modifying, or reengineering an object-oriented system requires an understanding of the object and class interactions which occur as a program executes. This work seeks to identify, visualize, and analyze interactions in object-oriented program executions as a means for examining and understanding dynamic behavior. We have discovered recurring interaction scenarios in program executions that can be used as abstractions in the understanding process, and have developed a means for identifying these interaction patterns. Our visualizations focus on supporting design recovery, validation, and reengineering tasks, and can be applied to both object-oriented and procedural programs.

Keywords

software visualization, object-oriented software engineering, program understanding, reverse engineering

UNDERSTANDING BEHAVIOR

We began our work with the hypothesis that visualizing interactions in object-oriented program executions can assist with software engineering tasks requiring program understanding. The importance of dynamics in the design, implementation, and modification of object-oriented (OO) systems cannot be over-emphasized. The communication dialog between classes and/or objects is typically designed using the notion of scenarios, often expressed using graphical notations such as event trace diagrams or interaction diagrams[24, 13]. Such diagrams are common in the design of procedural and parallel systems[3].

To truly understand how a component such as a class is used one must understand the scenarios within which that class' methods can and should be invoked. This behavioral information is not evident in object models including the class, much less in the source code defining the class. It is (or should we say, should be) described in event trace diagrams or interaction diagrams specified during system design. This information is required to use a class correctly and effectively, and must be taken into account when a system is modified.

Not only is it difficult to design these dynamic relationships, standard languages currently do not provide implementation support for interactions as first-class entities[1]. The gap in terms of abstraction between design-level behavioral models and the source code which implements a system can result in improper mappings from design to implementation. As a system evolves, modifications may cause a further drift from the documented (or intended) design.

We have created scalable visualizations to examine program event traces numbering in the hundreds of thousands. Interactive graphical visualizations can present this voluminous information much more effectively than textual representations, allowing a user to control the filtering and abstraction of available information. Using these views we have observed recurring sequences of interaction between classes and objects in OO systems. We hypothesize that high-level program behavior can be abstracted out from low-level event traces via these *interaction patterns*, and have developed a compact data structure which allows us to identify the existence of interaction patterns in large program event traces.

Citrin, et al. have emphasized the importance of tools which display and manipulate TMFDs to help document and explain a system's behavior, visualize complex trace data, and compare observed and predicted behavior[3]. We envision the use of our prototypes to support software engineering tasks in several ways:

- Identifying interaction patterns in program event traces can help an analyst construct design-level behavioral models from the low-level behavior of a system. This is useful during reverse engineering tasks, especially as support for reengineering legacy

[1] To meet this need, several research efforts have investigated language support for associations[8].

systems.

- Visualizations of the abstract behavior can be compared with design level information, such as execution scenarios or interaction diagrams, to help validate the design/implementation and to identify areas which need reengineering.
- Adding these capabilities to a CASE tool could help programmers keep behavioral models up to date with respect to modifications of a system's implementation, a common problem in practice[21].

Indeed, the aim of our visualizations is to facilitate design recovery, validation, and reengineering tasks by exposing dynamic interactions. While our prototypes were originally intended to be used on object-oriented systems, the call trace of a procedural program can be analyzed as well. The next section describes our focus on interactions as abstractions for program understanding and how we identify interaction patterns. Following that we discuss visualization tools that have been created, related work, and conclusions and future work.

FOCUS ON INTERACTIONS

For the purposes of clarity in the remaining discussion, a few formal definitions are in order. We define an *actor* as an entity in a system, having some "object-like" meaning. An actor might be a class, function, object, file, package, or thread. A grouping, or containment, hierarchy exists among these actors, which is a useful abstraction mechanism for analyzing interactions. For example, a file groups classes and functions, but not objects. A class contains all functions of that class and groups all objects of that class. A function groups all objects that implement that function. A particular set of actors can be grouped together as a component.

An *interaction* is a dynamic relation between actors. These are typically a message passed from one actor to another, the instantiation/deletion of a class or object actor by another actor, or the referencing of one actor's data by another actor. Our prototypes currently only instrument programs to track message interactions.

An *interaction scenario* is then a sequence of interactions between a set of actors that occurs as a program executes. One scenario can be interleaved with another. *Interaction patterns* are recurring interaction scenarios, manifested as repeated sequences of messages (message patterns) and/or recurring instantiation of objects (instantiation patterns). Like *design patterns*[10], interaction patterns exist at various levels of abstraction, from language constructs to system architectures. Note that our use of the word "pattern" is different than that of *design patterns* or *pattern languages* [10, 4], yet the two are related–see the section on Related Work. At a low level, interaction patterns can result from implementation aspects such as iteration through a linked-list data structure. They also exist at the design level, where they result from semantic operations such as class-uses-class associations. At a very high level, recurring interactions can be seen due to repeated usage of a system. While these definitions may seem object-oriented in nature, a procedural program can be characterized if a function or file is considered an actor and a function call is considered an interaction.

Identifying Interaction Patterns

In order to visualize and analyze large program executions a compact representation of the event trace and a way to extract the occurrences of interaction patterns is required. We have developed a compact representation of the call trace which can be used by tools to analyze large message traces. In a spectrum of possible representations of calling behavior that pit space overhead versus information accuracy, the call graph and the dynamic call trace represent two endpoints. At one extreme, a call graph is a compact representation of calling behavior that summarizes all possible run-time activation stacks. There is much interesting information about calling behavior that is dropped to gain compactness. The sequencing of calls, the context in which certain calls are made, conditional and indirect calls, and repeated calls are all examples of calling behavior that are lost. These problems exist in software tools that use the call graph to summarize dynamic program properties. For example, the inaccuracy of program profilers such as gprof [12] and qpt [19] can be traced to their use of the call graph to summarize context-dependent profile information in a context-independent manner.

At the other end of the spectrum, the dynamic call trace is an unbounded data structure containing a record of all the calls and returns that occur in a program's execution, regardless of whether the calls are direct or indirect. Extracting the call trace may incur high run-time overhead and storing the trace may not be feasible for long running programs. Furthermore, there is a data explosion problem: finding interesting information from the mass of data in the trace is not easy. Some trace-based tools animate the call graph to show the trace on the fly (without storing it) [2], or compute statistical summary information from the trace [6]. Both of these techniques deal with the space problem by ignoring or summarizing a large amount of dynamic information, as is done with the call graph.

We would like to have the best of both worlds: a compact representation (such as the call graph) that also retains as much information as possible about dynamic calling behavior (such as the dynamic call trace). We have developed a middle ground that allows a range of possibilities in this tradeoff. Our data structure also provides various abstract views of the dynamic infor-

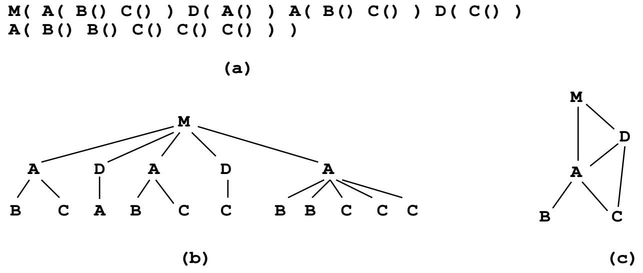

Figure 1: (a) A call trace, (b) its corresponding call tree, and (c) call graph. Edge directions are assumed to be directed down the page.

mation and serves well as a query engine for software tools dealing with calling behavior. One such abstract view of this data structure is the notion of a message pattern, as defined previously.

Implementation Details
There are three basic ideas we use to compact the dynamic call tree. First, we use hash consing to ensure that identical tree structures in the dynamic call trace are represented exactly once in the compact representation. Second, we compactly summarize repetitive sequences of subtrees that are generated by loops. These sequences can be summarized at varying degrees of accuracy, resulting in different compact representations (and subsequently different levels of message pattern abstractions). Finally, we compress repetitive calling chains that are generated by recursion. The compact representation of the dynamic call tree is a directed acyclic graph (`dag`).

Figure 1 shows an example call trace, call tree, and call graph. Our compact representation of the call tree from Figure 1(b) is shown in Figure 2. Each vertex corresponds to a call. It is clear that this representation captures exactly the same information as the call tree.

The basic framework for parsing a call trace to produce a `dag` is straightforward. The analysis requires three main data structures: a stack of active procedures, a heap of `dag` structures, and a hash table for determining if a particular `dag` structure has been built already. The `dag` structures are built in a bottom-up fashion (from the leaves of the dynamic call tree to the root). Hash consing ensures that if a tree data structure is constructed bottom-up, then duplicate trees will always hash to the same element.

Hash consing results in the sharing of subtrees in the `dag`, as is evident by the shared subtree of A calling B and C in Figure 2. This subtree is a message pattern, because it has more than one incoming parent edge. We have implemented a pattern iterator that walks the `dag` and returns message patterns that are encountered. In addition to looking for shared subtrees, the pattern iterator also looks for repetitive subtrees sequences that resulted from iteration in the program execution.

Discussion
The pattern recognition algorithm that we have described does have its limitations. Identifying duplicate subtrees in the call tree will not find repeated sequences of subtrees. For example, suppose that procedure A calls procedure B and then calls procedure C, and then repeats this calling pattern. The pattern "B followed by C" will not be detected because B and C are separate subtrees in the call tree. We extended our algorithm to detect such patterns using well-known substring matching heuristics[1]. Still, as human thinkers we can use our visualization prototypes to find more *visual* patterns in a message trace than our pattern iterator finds. However, the pattern detection capabilities are extremely useful as a starting point for the human pattern-matching process.

By limiting the stack depth of our pattern identification in the `dag` representing the call trace, we can filter out lower-level behavior and get very high-level patterns. Additionally, repeated patterns can be represented as a single instance as the dag is traversed, or a logarithmic reduction in the number of repetitions can be done. These facilities allow our prototypes to filter and abstract the voluminous information which is present in program execution traces.

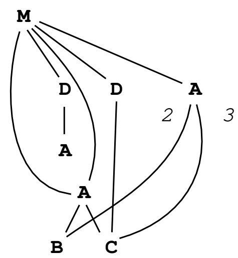

Figure 2: A compact representation of the call tree from Figure 1b.

VISUALIZATIONS

Our work on visualizing the dynamics of object-oriented systems began with the idea that it might be useful to expose the interaction between objects as an OO system executes, and has evolved to use visualizations of interaction patterns to support program understanding tasks. The first visualization prototype we developed animated the instantiation of objects and the message passing between objects in a single view. The visualization was based on call trace files generated from `C++` source code annotated by hand. Only small, toy programs were visualized. The next generation of visualization used multiple views to show the call stack, inheritance hierarchy, instances, and message passing[14]. Animation was still used to show the progression of time, but larger programs were visualized.

The next sub-sections include a summary, impact, and descriptive example for two of our current visualization prototypes, which we will refer to as **P1** and **P2**. Readers not wishing to understand the details of the prototypes should skip the example sections. Both tools can be used to visualize the execution of `C++` or `C` programs. The visualizations are written in `C++`, `X Windows`, and `Motif`.

Prototype P1

Summary

The first prototype allows a user to display and browse real-world sized event traces (100,000+ messages). These traces are generated by executing a subject program instrumented by automatically inserting tracing objects into the source code. The **Execution Mural** view (see Figure 3) shows time on the horizontal axis and messages passed between classes in a program on the vertical (much like an event trace diagram rotated 90 degrees). We take the approach of creating a general view initially showing all the classes and messages, and providing several visual filtering mechanisms which allow a user to focus on information of interest. These include a global navigational Information Mural[15] which portrays the entire trace, the ability to change the ordering of classes along the vertical axis, selectively show or hide particular classes, color-code specific messages, and zoom in on sub-sections of the message trace.

Any visualization which supports program understanding tasks must include a view of the source code for the system being examined, since that code is the concrete realization of the system (unless of course, the system is programmed visually). While different abstract models of the system can be developed to help a person understand its behavior, if that person will be modifying the system the abstractions must eventually be related to the source code. In P1, a **Code View** (see Figure 5) depicts the messages between classes and functions overlaid on an abstract view of the source code. This helps relate the messages in the Execution Mural to the source code from which the event trace is generated.

Impact

With P1 we found that animating the execution of a program is less important for global program understanding tasks than being able to browse the entire execution and focus on areas of interest as needed to undercover specifics. We also narrowed the focus of our work to the sequence of object interactions, while others published work focused on cumulative dynamics[5, 6]. Scalability became the number one issue in creating useful visualizations to aid the understanding processes during implementation and maintenance of real-world sized systems, with the Information Mural technique created to support this goal[15].

The P1 prototype allowed us to visually examine event traces of several different programs, including Polka[25] and SeeSoft[9]. Visual patterns can be seen in an entire message trace, and then lower-level patterns as we zoom in on sub-sequences of the execution. The visual patterns are either the result of similar semantic operations in the code or of iteration as in a `for` loop. One of the weaknesses of the visualization in terms of helping program understanding tasks is that a view of individual messages is really too low-level compared to a user's mental model or system design models such as interaction diagrams. The message patterns we were finding seemed to be useful abstractions to help bridge this gap. The work with the P1 prototype thus motivated the development of the compact message trace representation and the views implemented in P2.

Example

The process of using P1 includes several steps: 1) static analysis of the source code using `gen++`[7], 2) automatic annotation of source code by a `Perl` script which places tracing objects in the code as described by O'Riordan[23], 3) compilation and execution of the annotated source to generate dynamic event trace files, and 4) visualization of the information in the trace files. Two Execution Mural views from the P1 prototype are shown in Figures 3 and 4. The subject program is a Polka[25] bubble-sort algorithm animation. Polka is an object-oriented toolkit for creating algorithm and program visualizations, written in `C++`.

The major visual innovation in the Execution Mural is the ability to create a global overview of a message trace containing hundreds of thousands of messages. The technique utilizes grayscale and color shading along with anti-aliasing techniques to create a miniature representation of an entire large information space. Such a view is called an *Information Mural*; the technique is described in [15, 16].

The importance of this global overview quickly becomes apparent as one uses P1. First of all, viewing an entire execution trace gives an analyse immediate insight into different phases of the execution (see Figure 3). Murals of program traces are very repetitive, even as the mouse is used to zoom in on smaller "windows" of the trace. This observation led us to develop the automatic pattern detection features mentioned previously. Second, locating particular messages in a real trace is nearly impossible without some search mechanism. The global overview along with message coloring facilities allow the trace to be searched visually see Figure 4).

The Code View in prototype P1 (Figure 5) uses an abstract representation of a file as a collection of class and function definitions. Files are listed horizontally across the view, with the classes in each file represented as

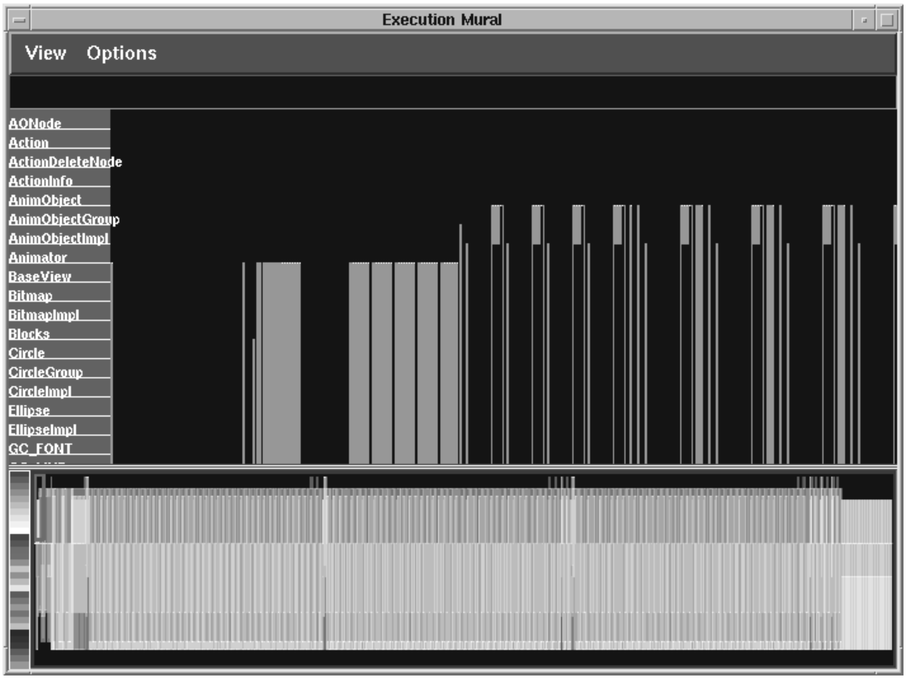

Figure 3: Initial P1 Execution Mural of a Polka bubble-sort. Classes are on the vertical axis, and each message is single pixel wide vertical line from source to destination class. The global view at the bottom is a mural of the entire set of over 52,000 messages; a navigation rectangle shows the position of the focus area relative to the rest of the classes and messages in the trace.

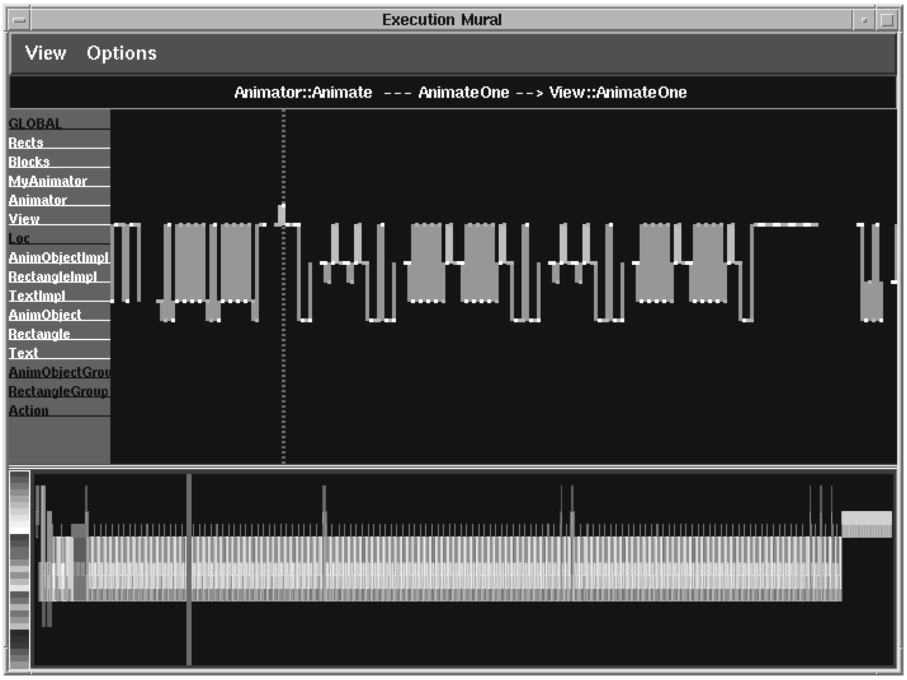

Figure 4: Execution Mural from Figure 3 with several classes hidden, the message width increased, and several messages highlighted in different colors.

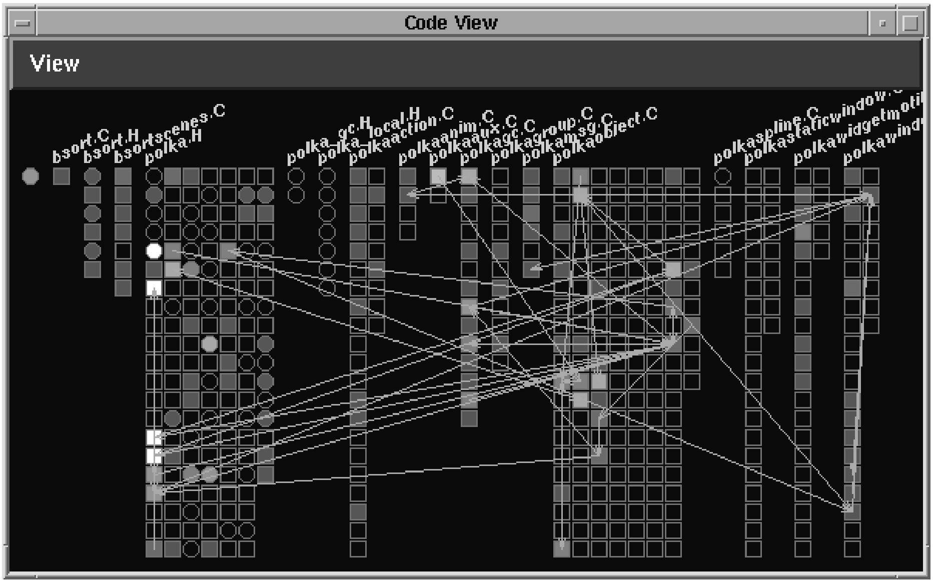

Figure 5: P1 source Code View of Polka animation toolkit, with files arranged horizontally. Circles represent class definitions and squares represent functions; the fill color shows relative message-passing volume for each node for a particular execution trace. Arrows indicate a set of messages that were highlighted by the user in the Execution Mural view.

circles and functions in each file as squares. Dynamic information gathered from the event trace is overlaid in this view by coloring the nodes according to the relative message-passing volume for each node, with the most active being white. When the mouse pointer brushes over a node, its name appears and members of its class are highlighted. When the user highlights messages in the Execution Mural view, arrows appear connecting the corresponding function nodes in the Code View, as shown in Figure 5. Notice in Figure 5 that more control-oriented functions appear to be sources of many edges, while computational or access functions look like sinks.

Prototype P2

Summary

The second visualization prototype (P2) is focused on visualizing interaction patterns to support design recovery and reengineering tasks. It incorporates a similar view to P1's Execution Mural, with the addition of the automatic message pattern detection methods described in a previous section and several message pattern-oriented views. Patterns the tool can detect are used as a starting point for presenting the user with interaction patterns in execution traces. The visual interface then allows the user to examine the message patterns and look for new ones at various levels of abstraction.

P2 allows us to create views of a particular program execution based on static information about the program and a trace file of interesting events (function calls and returns). The views are *Observers*[10] of a single program model which contains both static and dynamic information, and they co-exist in a single Viewspace window which acts as a *Controller*[11] to handle user input. A *Composite*[10] class hierarchy defines views as visual objects themselves. Interaction occurs through pointing with the mouse and using pop-up menus which are associated with the various views.

Impact

From usage scenarios for the P2 prototype like the one we describe in the next section, the following can be concluded: 1) interaction patterns exist in program executions, 2) we can automatically detect them, and 3) they are at a similar level of abstraction as design level models in system design documents or programmer's mental models. It has become clear that understanding the behavior of a few interaction patterns can go a long way toward understanding the entire execution scenario; in the P2 example, variations of a single pattern constitute over 80 percent of the messages in the trace. While this will obviously not always be the case, it is clear that interaction patterns can be used as abstractions which relate the low-level implementation to higher level design models during design validation and recovery tasks.

Example

An example usage scenario will serve nicely to present the features and usefulness of P2. We describe using the tool to examine the Polka program animation toolkit mentioned previously. We are interested in comparing the Polka toolkit designer's mental model of its behavior with the actual implementation. The specific focus in this scenario is on the interactions taking place as each animation frame is rendered by Polka. A software engineering task such as this might occur during validation of an implementation with respect to system design documents, or during construction of design models while reverse engineering a system architecture.

Figure 6 shows an event trace diagram made by the Polka designer to describe the interactions involved in Polka while animating a frame. A trace file for a Polka bubble-sort algorithm animation which consists of almost 64,000 function invocations is read and processed by P2. We first create a global **Execution Mural** (Figure 7a) of the entire message trace. This view will act as a global overview, showing where the message patterns that are identified fit within the execution. Notice that the Execution Mural views in this prototype are slightly different from the previous generation in P1. The view has been rotated to look more like interaction diagrams, with the 40 classes in the program on the horizontal axis and the almost 64,000 messages drawn as horizontal lines down the vertical axis using the Information Mural compression technique[15]. Areas that are brighter in the mural are more dense with information, conveying the same visual patterns that would be apparent if a huge event trace diagram of the entire program was observed from a distance. This global Execution Mural does not have a focus area, it just shows all of the messages at once.

Notice how repetitive the diagram is visually. A distinct pattern appears in the beginning, followed by another that repeats six times. To get a feel for the information compression ratio, Figure 7b shows approximately the first 10,000 function invocations in the trace. In part (b) more visual patterns become apparent as we zoom in to a finer resolution.

Now we create a **Pattern Matrix** (Figure 8) showing the classes involved in the top-level message patterns that were identified by our pattern detection algorithm described previously. "Top-level" means the largest sequence of messages that occur more than once and begin closer to the root of the message dag than other subsequences that might also be message patterns. The matrix assigns a message pattern to each column; message patterns are identified by the first message name along with the global message number of the first message in the pattern. The rows of the matrix correspond to the classes in the program. The matrix is created using the Information Mural technique as well, so is effectively scalable to hundreds of classes and patterns. Note that the order of message patterns along the horizontal axis can be changed to group patterns by name, size, or order of occurrence in the trace. A Pattern Matrix mapping functions to patterns is also available.

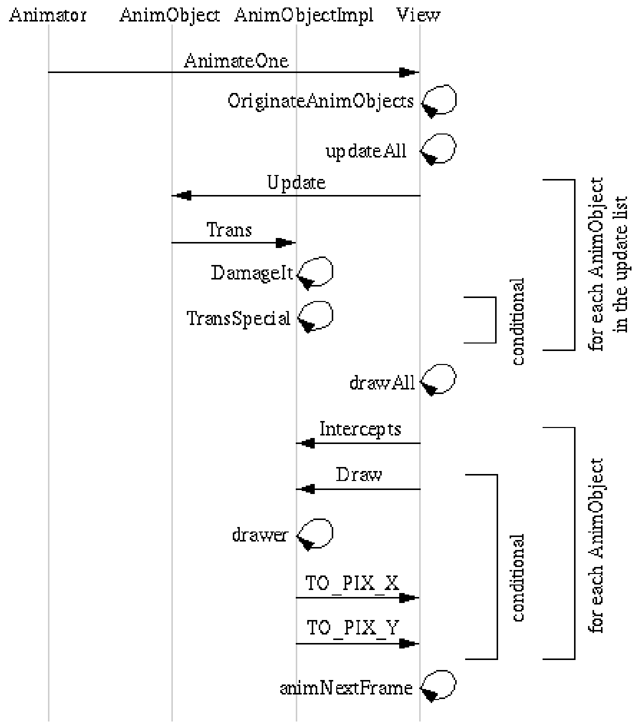

Figure 6: Designer's event trace diagram of process to animate one frame in the Polka animation toolkit.

(a) (b)

Figure 7: (a) Global Execution Mural for the Polka bubble-sort animation, essentially a miniature event trace diagram of the entire message trace. The vertical resolution is 64,000 messages on 400 pixels, an information compression ratio of 160:1. (b) Part (a) zoomed in on approximately the first 10,000 messages of the trace. The information compression ratio is around 25:1.

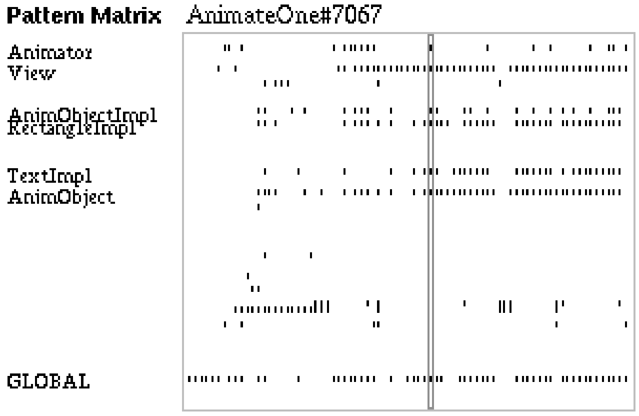

Figure 8: Pattern Matrix for the Polka bubble-sort animation. Message patterns are assigned columns in the matrix, and rows are classes in the program. Entries are made for classes which are "members" of each message pattern. Pattern AnimateOne#7067 is currently highlighted.

From this view we can see which patterns might be related to the designer's execution scenario by looking for ones which contain the same classes. The designer's diagram (Figure 6) includes the `Animator`, `View`, `AnimObject`, and `AnimObjectImpl` classes. The Pattern Matrix has several `AnimateOne` patterns which contain most of these classes, for example Figure 8 has the `AnimateOne#7067` pattern highlighted, which contains classes `Animator`, `View`, `AnimObjectImpl`, `RectangleImpl`, `TextImpl`, `AnimObject`, and the `GLOBAL` class which represents functions in the global scope. This message pattern is also a likely candidate because its first message is `AnimateOne`, which is same as the first one in the designer's diagram.

The **Pattern Mural** view (Figure 9) gives a time ordering to the message patterns shown in the matrix by showing message patterns on the vertical axis, and where they occur in the program execution along the horizontal. This view uses the Information Mural technique by drawing a point for each message in the execution, at sequential x coordinates and at the appropriate y coordinate for the message pattern to which that message belongs. Note that "sequential x coordinates" are in terms of the message order, not the pixels on the screen: many messages may be compacted into the same column of pixels.

The order of patterns along the vertical axis can be changed as in the Pattern Matrix view; Figure 9 shows patterns in order of occurrence (first at the top). In this view we notice several distinct `AnimateOne` patterns which occur in the middle of the trace. We hypothesize that each of these patterns corresponds to the distinct phases in the global Execution Mural of Figure 7a. If

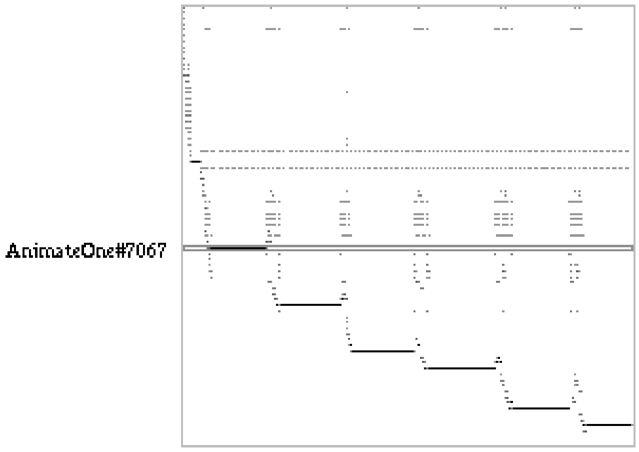

Figure 9: Pattern Mural for the Polka bubble-sort animation. Message patterns are assigned to the vertical axis by order of occurrence; a point in the mural is made for each message at sequential x coordinates, with the y value corresponding to the pattern of which that message is a member (messages that are not members of patterns are not shown). The AnimateOne#7067 message pattern is highlighted.

we turn on highlighting of the current selected pattern in the global Execution Mural, we confirm this suspicion. Note that all the views are synchronized so that as we change the current pattern in one view the others change to show the location of that pattern as well. Our technique of providing multiple views of the information we are analyzing allows us to make observations such as this.

The `AnimateOne` pattern seems be the one we are looking for to compare with the designer's mental model of frame animation. We now use our system to create an Execution Mural of the `AnimateOne#7067` pattern, shown in Figure 10. The mural on the right hand side provides a global overview of all the messages in the pattern, and acts as a two-dimensional scroll bar for moving the focus area on the left. Messages corresponding to both function calls and returns can be displayed; calls are solid and returns are grayed. Horizontal lines represent messages, and name labels can optionally be displayed above each message. A circle marks the destination end of the message. Because the designer's event trace diagram does not include global function calls (they are mostly for the graphics), we can remove the `GLOBAL` class by using the mouse to select the class label and choosing a menu option to remove that class. Another menu option allows us to eliminate the return messages from the display. We can scroll the Execution Mural of Figure 10 and compare it with Figure 6 to see how the implementation behavior conforms to the expected design.

From a quick glance through the entire pattern, we see good conformance to the expected behavior. There are four `Update`s, two which deal with `RectangleImpl`s and two which deal with `TextImpl`. The designer did not include these classes in his diagram, presumably because they are specializations of the `AnimObjectImpl` class. Because it is common for design models to deal with abstract classes, a feature we are adding to the Execution Mural will allow subclass behavior to be generalized into the base class.

The designer concludes that in this frame of the bubble-sort animation two bars with their text labels are changing places. There are some messages (`BoundBox`, `DamageCheck`) which are not in the designer's diagram, but are in the correct place according to the designer. Note that the original diagram only had one `DamageIt` message after the `Trans` message, where the observed pattern has two. The designer confirms that there should be two, because one is for the old position of the object and one is for the new position after the object moves, changes size, or does some other action. Here is a case where the design model would need to be updated.

For this example, the `AnimateOne#7067` pattern does in fact appear to implement the frame animating process as expected. After investigating Execution Murals of other `AnimateOne` patterns, it seems that the differences between them result from `AnimObject`s first originating in particular frames or different `AnimObject`s being updated and drawn. Our visualizations should make it easy to uncover these differences by providing visual "diffs" of patterns, a feature we are adding.

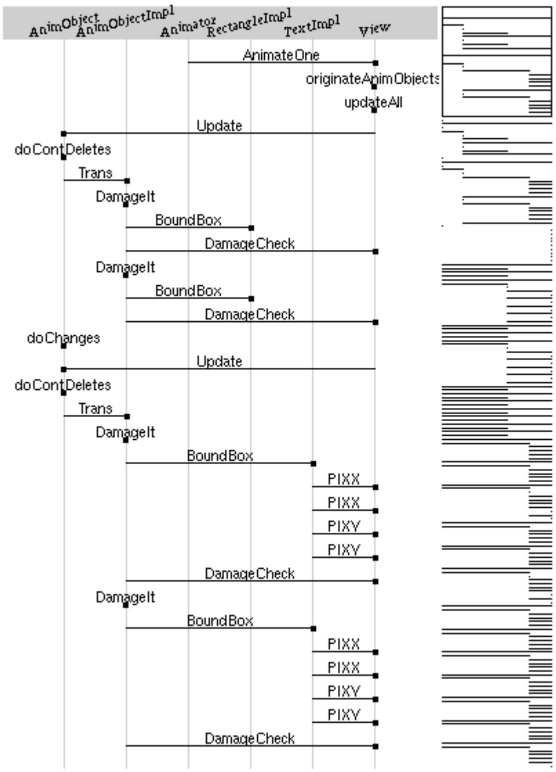

Figure 10: Execution Mural from prototype P2 of message pattern AnimateOne#7067. The mural on the right gives a global view of all messages in the pattern and acts as a two-dimensional scrollbar for the focus area. The navigation rectangle in the upper part of the global view corresponds to the messages displayed in the focus area. The GLOBAL class has been removed and only messages corresponding to function calls are shown.

RELATED WORK

Several different areas overlap with our work, including software visualization, program understanding, reverse engineering, and OO methods. Some of the more recent efforts in these areas are mentioned here and related to our work.

As mentioned previously, Citrin, et al. have attempted to formalize the notations used to describe communication between entities in systems, using the notion of a temporal message-flow diagram (TMFD)[3]. They have built tools to display and edit TMFDs, to generate TMFDs from event traces, and to simulate the operation a system using TMFDs. Their work is much more general than ours, handling systems in which messages can be sent and received in an interleaved, non-deterministic sequence. However, they have not done any work to identify patterns in the event traces.

The notion of a pattern as a solution to a problem in a particular context provides a literary form through which experience with software can be documented to be reused by others[10, 4]. In contrast, our interaction patterns are so named because they too are a repeatable entity and because they create visual "patterns" on the screen when we visualize OO message traces. The relation between the two types of patterns arises in that interaction patterns will result from various design patterns, and can be seen as low-level evidence for the existence of a design pattern. In this way, identifying message patterns can be seen as a form of "design pattern mining."

The Program Explorer is a `C++` program understanding tool that is focused on class and object centered views[18]. The authors have developed a system for tracking function invocation, object instantiation, and attribute access. The views show class and instance relationships (usually focused on a particular instance or class), and short method invocation histories. It seems that the system is designed to execute the program for a while, stop execution, and then focus in on particular classes or objects. It's not intended as a global understanding tool, so the user must know what (or where in the execution) they are interested in beforehand. Examples of using the system to uncover design patterns in real-world sized systems are given. Again, it seems that the user must know the design pattern and have an idea where that pattern occurs to exploit the visualizations.

HotWired is a visual debugger for `C++` and `Smalltalk` that provides both standard object views and a scripting language to create simple program visualizations[17]. Views show instances of classes (similar to [5]), message passing between individual instances, and instance attribute values. It is possible to "record" particular message traces to be replayed. Their recording strip view shows instance activation over time, and could benefit from our Information Mural technique. The visualizations focus on debugging tasks, and to support custom debugging a scripting language maps instance values to visual objects. However, these scripts must be written manually.

De Pauw, Helm, Kimelman, and Vlissides[5, 6] have developed visualization techniques and a tool for presenting attributes of object-oriented systems. The authors use portable instrumentation techniques to extract the required information about a program's execution. They also developed views, most of which are chart-like, that present summary information about the execution. These views are quite effective for analyzing program performance and class relationships in terms of the amount of interaction between classes and objects. However, the information they capture is mostly post-mortem summary information, whereas we seek to uncover the semantics and sequence of the interactions. The authors made this compromise when they decided not to store incremental information about the execution in favor of storing more cumulative information.

The OO!CARE tool is the `C++` version of the CARE environment for `C` program understanding[20]. The idea of the OO!CARE system is to extract and visualize dependencies between classes, objects, and methods in the program, as well as the control and data flow. The system includes a code analyzer, a dependencies database, and a display manager. The hierarchically designed views present class inheritance, control-flow dependencies, and file dependencies. A column oriented view called a *collonade* presents data-flow dependencies. The dependencies are extracted statically, so in the case of a virtual function call in C++ a "dummy" member function is created to represent all the possible run-time bindings. While the views provide zooming and panning capabilities, plus hierarchical decomposition, the examples given do not demonstrate that they scale to handle large programs.

Murphy, et al. have developed an approach that allows software engineers specify high-level models of a system and how the source code maps into that model[22]. Then a *reflexion model* is computed which uses call graph and data referencing information to determine where the model agrees and disagrees with the actual implementation. A box-and-arrow type diagram is used to depict the specified models and their differences. Their approach has helped with design reengineering and conformance tasks. This work is directed more toward static, architectural models, while our work is focused on more sequential, behavioral type models.

CONCLUSIONS AND FUTURE WORK

We have described several visualization prototypes which reveal the interactions that take place as a program executes. The first utilizes innovative global overviews of large program event traces, and allows the information to be filtered and highlighted to uncover details. The second prototype embodies the notion of interaction patterns as higher-level abstractions that can be compared with design level execution scenarios. The usage example for the P2 prototype described in this paper gives evidence that interaction patterns exist, we can automatically detect them, and that they are at a similar level of abstraction as design level models in system design documents or programmer's mental models.

Software visualizations of any kind are only useful if they scale to handle real-world systems. We have discussed some techniques for storing and presenting large program event traces, and different alternatives which vary the level of abstraction reflected by the message patterns. For example, when we ignore multiple iteration in the call trace or limit the stack depth we get "higher-level" message trace summaries which might be more useful for global understanding. Being able to interactively control this filtering and abstraction is a new feature we are adding to our system.

As is clear from our discussion in the second section, we characterize interactions as messages between actors or the instantiation/deletion of an object actor by another actor. The current trace analysis prototypes track only message patterns; the ability to trace object instantiation and destruction is currently being implemented, giving rise to more complete interaction scenario analyses. Additionally, the actors in the current interaction patterns are files, classes, or functions; it would be useful to abstract down to the object level.

The P1 and P2 prototypes are currently the basis for a new tool, the Interaction Scenario Visualizer (ISVis), being used to validate reverse-engineered architectural models in research seeking to assist with the evolution of legacy systems. The purpose of the project is to develop analysis techniques which can predict how a legacy architecture needs to change in response to changing requirements. However, before that analysis can take place a correct model of the legacy architecture must be established. Usage scenarios are first used to describe and test the existing system, and event traces are generated. The new prototype allows a user to visualize interaction scenarios, identify and understand patterns of interaction in the scenarios, input design-level models to compare with recovered scenarios, and store analysis sessions for later use. Many of these features make the new prototype more usable, especially because it allows a user to build and save high-level models of behavior. The pilot system for this study is NCSA's Mosaic web browser.

REFERENCES

[1] Aho, Hopcraft, and Ullman. *The Design and Analysis of Compuer Algorithms*. Addison-Wesley, New York, 1974.

[2] H.-D. Bocker and J. Herczeg. What tracers are made of. In *Proceedings of the ECOOP/OOPSLA '90 Conference*, pages 89–99, Ottawa, Ontario, Oct. 1990.

[3] W. Citrin, A. Cockburn, J. von Kanel, and R. Hauser. Using formalized temporal message-flow diagrams. *Software Practice and Experience*, 25:1367–1401, 1995.

[4] J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[5] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the ACM OOPSLA '93 Conference*, pages 326–37, Washington, D.C., Oct. 1993.

[6] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In *Proceedings of the European Conference on Object-Oriented Programming '94*, 1994.

[7] P. Devanbu. A language and front-end independent code analyzer. In *Proceedings of the International Conference on Software Engineering*, Australia, May 1992.

[8] S. Ducasse, M. Blay-Fornarino, and A. M. Pinna-Dery. A reflective model for first class dependencies. In *Proceedings of ACM OOPSLA '95*, pages 265–280, 1995.

[9] S. G. Eick, J. L. Steffen, and E. E. S. Jr. SeeSoft—A tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, Nov. 1992.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Pattterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[11] A. Goldberg. *Smalltalk-80, The Interactive Programming Environment.* Addison-Wesley, Reading, PA, 1983.

[12] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software Practice and Experience*, 13:671–685, 1983.

[13] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, Reading, MA, 1992.

[14] D. F. Jerding and J. T. Stasko. Using visualization to foster object-oriented program understanding. Technical Report GIT-GVU-94-33, Georgia Institute of Technology, July 1994.

[15] D. F. Jerding and J. T. Stasko. The Information Mural: A technique for displaying and navigating large information spaces. In *Proceedings of the IEEE Visualization '95 Symposium on Information Visualization*, pages 43–50, Atlanta, GA, October 1995.

[16] D. F. Jerding and J. T. Stasko. Using Information Murals in visualization applications. In *Proceedings of the 1995 Symposium on User Interface Software and Technology (Demonstration)*, pages 73–74, Pittsburgh, PA, November 1995.

[17] C. Laffra and A. Malhotra. Hotwired – a visual debugger for C++. In *Proceedings of the USENIX 6th C++ Technical Conference*, April 1994.

[18] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of ACM OOPSLA '95*, pages 342–357, 1995.

[19] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. Technical Report Computer Sciences Technical Report 1083, University of Wisconsin-Madison, 1992.

[20] P. K. Linos and V. Courtois. A tool for understanding object-oriented program dependencies. In *Proceedings of the Workshop on Program Comprehension*, pages 20–27, Nov 1994.

[21] R. Malan, D. Coleman, and R. Letsinger. Lessons from the experiences of leading-edge object technology projects in Hewlett-Packard. In *Proceedings of ACM OOPSLA '95*, pages 33–46, 1995.

[22] G. C. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the gap between source and high-level models. In *Proceedings of the Foundations of Software Engineering*, page ??, 1995.

[23] M. J. O'Riordan. Debugging and instrumentation of c++ programs. In *Proceedings of the USENIX C++ Conference*, pages 227–242, Denver, CO, Oct 1988.

[24] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, New York, NY, 1991.

[25] J. T. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.