Ph.D. Defense of Research Summary

Visualization of Interaction Patterns in Program Executions

Dean F. Jerding
February 23, 1996

Abstract

Implementing, validating, modifying, or reengineering a software system requires an understanding of the program's behavior. Several factors make this difficult, including the gulf in terms of abstraction between design models and source code, and the design drift that occurs over the lifetime of a system. This work seeks to identify, visualize, and analyze interactions in program executions as a means for examining and understanding behavior. Recurring interaction scenarios in program executions have been discovered, which can be used as abstractions in the understanding process. Tools have been built to support the process, utilizing innovative visualization techniques and a means for identifying these interaction patterns. The hypothesis is that visualizing interaction patterns in program executions can facilitate program behavioral understanding during design recovery, design/implementation validation, and reengineering tasks. Several case studies have been performed to validate this hypothesis.

Keywords: software visualization, program understanding, reverse engineering, program behavior, dynamic program analysis

Table of Contents

MOTIVATION **1**

APPROACH **1**

DEFINITIONS 1
FUNDAMENTAL HYPOTHESES 2

TOOLS **2**

VIZBUG++ 2
VOODOO 3
PEASOUP 4
V4.0 4
ISVIS 5

CASE STUDIES **6**

CASE I: REENGINEERING VIEW REDRAW MECHANISMS IN GUI APPLICATIONS 6
CASE II: DESIGN RECOVERY OF MOSAIC WWW BROWSER 11

CONCLUSION **15**

REFERENCES **15**

Motivation

The underlying motivation for this work is the difficulty and lack of support for understanding the behavior of software systems. A correct dynamic model of a program is required for many software engineering tasks, including design, implementation, testing, maintenance, and reverse engineering. This research focuses in particular on design recovery, design and implementation validation, and reengineering tasks.

Difficulties in understanding the dynamics of a program arise for several reasons. One of the major problems is the gulf in terms of abstraction between design level models and the source code which constitutes the implementation. Part of this gap is due to the level at which humans try to solve complex problems versus the level at which computers are currently programmed. Additionally, while most standard programming languages (especially object-oriented) support the transition of static design models into code, there is no direct language support for specifying behavioral protocols.

Not only does the gulf of abstraction lead to inconsistencies between intended design and the implemented system, but often the design of a system is changed as it is implemented. This design drift often goes undocumented, especially if functional requirements are not affected by the changes. As systems evolve, with new features being added and bugs fixed, more versions are produced. Soon the only accurate documentation is the source code. Safe and efficient changes to the system cannot be made without correct behavioral models of the existing architecture and design.

Visual techniques and tools seem useful to help a human understand such a large and complex information space. In the analysis and design phases of software development, graphical notations and diagrams are often used to specify both static and dynamic aspects of a system. Such visual representations are valuable because they convey information to the observer more efficiently than would purely textual descriptions. In the implementation stage, these diagrams along with related textual specifications must be translated into the source code. Tools which then analyze the source code and the corresponding system should present information in visual forms corresponding to those with which the system was designed. In order to accomplish this, mechanisms must exist to abstract and filter voluminous low-level information in the implementation up to design levels. Current visualization tools do not provide such mechanisms and thus do not scale well to handle the large amounts of information involved when analyzing “real-world” sized systems.

Approach

This work began with the hypothesis that visualizing interactions in object-oriented program executions can assist with software engineering tasks requiring program understanding. The interactions are especially important because the behavior of OO systems seems to be more difficult to understand than static information about class and object data, and because the behavior cannot typically be determined until run-time. Through several visualization prototypes we have observed that execution traces of object-oriented programs are made up of repeated sequences and sub-sequences of messages. While we were initially focusing on the object-oriented paradigm, the notion of interactions between functions and components is useful for understanding behavior in the procedural paradigm as well. The identification and analysis of recurring interactions in program execution scenarios is the focus of the current visualization system.

Definitions

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. To allow for more arbitrary grouping of actors, we define a *component* to be any set of actors.

An *interaction* is a dynamic relation between actors, directed from one actor to another (possibly the same) actor. 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.

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). Interaction patterns can also be viewed as regular expressions over the interaction alphabet, used for pattern matching in scenarios. At a low level, these 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, procedural program concepts can be characterized as a subset of those actors and interactions defined above.

Fundamental Hypotheses

Several fundamental ideas are at the core of this research, guiding the approach to solving the problems identified in the first section:

Visualization of large information spaces

The use of graphical techniques to depict information on a computer display has proven a useful alternative for presenting various forms of information, including computer programs. We believe that visual representations of the voluminous information which can be derived from program executions are a necessary means for that information to be processed and analyzed. While off-line analysis of the data is obviously useful, it is hypothesized that visualizations supplementing the human pattern recognition and abstraction capabilities can yield the most productive results in such a complex process. The visualizations, however, will be useless to a human analyst unless effective representation and navigation techniques for large information spaces are developed.

Interaction patterns bridge gap between design/implementation

During the course of this research, it has been observed that program executions are made up of recurring interaction scenarios. As mentioned previously, these interaction patterns occur at various levels of abstraction. We hypothesize that these patterns can help bridge the gulf of abstraction between low-level execution events such as function calls, and high-level design models of program behavior. Humans typically solve complex problems using divide-and-conquer strategies, by observing patterns, and by finding analogies; interaction patterns can be utilized in all three of these processes.

Tools

Several generations of visualization prototypes have been built as part of this research, to depict the behavior of object-oriented and/or procedural programs. This section summarizes each of the prototypes and describes their impact on this research.

VizBug++

Summary

The VizBug++ prototype provides a basic view of an executing program. The view contains three basic entities: a tree structure representing the class hierarchy, rectangular nodes representing global functions, and circular nodes representing instances. The view is intended to graphically animate the message-passing that occurs during execution of object-oriented programs.

A library of visualization functions exists to construct and animate the view appropriately for each type of event. Simple C++ programs can be hand-annotated with these calls at places where interesting events occur in the program. Execution of the annotated source program then produces calls into the visualization library and causes the view to be updated. A simple user interface provides both continuous and discrete (step mode) control of the event visualization.

By stepping through the visualization of the program execution, users can observe existing objects and messages. User's can browse the relationships between existing entities by pointing and clicking. The resulting information can be useful toward understanding the dynamics of the program execution, thereby benefiting the program development and maintenance process.

Impact

From the results of this view and our experience implementing VizBug++, it is clear that gathering the necessary information to construct useful visualizations is indeed a difficult problem. View layout and information overload are major problems encountered in this simple view. Presenting the information in an organized and informative way seems to require multiple coordinated views, with different levels of abstraction. Given the role that object-oriented programming has begun to play in the software development arena, these problems merit further exploration.

VooDoo

Summary

This prototype involves the development of a framework for visualizing object-oriented program executions, extending the ideas embedded in VizBug++. The work on this problem can be divided into two basic areas: program event generation and program event visualization. Users can perform visualization of the source program either continuously or in discrete steps. When the Visualization Manager is ready for the next event, it sends a request to the Event Manager. After the requested event is received from the Event Manager, the Visualization Manager must then pass the event information to the underlying program state model and the visualization models specific to each particular view. In order to generate the views, we have done hand annotation of a simple C++ program. Executing the annotated source produces an event stream upon which our views are built.

The VooDoo framework includes four different views. The object view is a macroscopic view of all current instances in the program. Objects are shown at a pixel-like scale. Selecting an object allows the user to see information such as the object's address, name, and inheritance hierarchy. Additionally, the view might shows a history of active objects as the program executes. A view of the inheritance hierarchy displays the class inheritance structure for the program as a tree. Classes are shown as nodes labeled with their class name. Inheritance is shown by arrows from parent (base) classes to their children. The third view is a class-uses-class dependency matrix. Classes are listed in alphabetical order along each axis. The vertical axis shows classes who are dependent on classes on the horizontal axis. The last view depicts the call stack.

Impact

The event generation and event visualization capabilities of this prototype have revealed several problems and given rise to several questions:

- How much of this event information can be generated automatically, in an unintrusive manner?
- How do we permit two-way navigation through the program event stream, thus allowing programmers to rapidly focus on points of interest?
- What mechanism can we use for saving the state information needed to restore previous program execution states?

- How should the information models be defined in order to facilitate view synchronization, inter-view communication, and the addition of new views?
- How do we create views that scale well for large applications? Do new methods of abstraction need to be developed?

Answering these questions motivated the development of the Peasoup prototype.

Peasoup

Summary

The Peasoup 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 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[6] 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 Peasoup, a Code View 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 Peasoup 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[1,2]. 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[6].

What this view has allowed us to do is visually examine event traces of several different programs, including Polka[7] and SeeSoft[3]. 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 Peasoup prototype thus motivated the development of the compact message trace representation and the views implemented in the next generation prototype.

v4.0

Summary

The fourth visualization prototype (v4.0) is focused on visualizing interaction patterns to support design recovery and reengineering tasks. It incorporates a similar view to Peasoup'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 at various levels of abstraction.

v4.0 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* [4] 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* [5] to handle user input. A *Composite* [4] 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 of the v4.0 prototype, 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. 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.

ISVis

Summary

The purpose of this system is to support the browsing and analysis of execution scenarios, derived from actual program executions. It is useful during software engineering tasks requiring a behavioral understanding of programs, such as design recovery, architecture localization, design/implementation validation, and reengineering. The key features of ISVis are its use of visualization techniques to depict the large amounts of information available to a user, and the notion of recurring scenarios, or "interaction patterns", as abstractions which help bridge the gap between low-level event traces and high-level design models. Users of the system might be software designers, programmers, testers, or maintainers.

The process of using ISVis requires several phases:

1) static analysis of the subject program
2) instrumentation of the source code to track interesting events
3) usage of the instrumented system in particular scenarios to generate program event traces
4) analysis of the event traces using the ISVis graphical views
5) recording of models created by the user

While this system is intended to handle object-oriented (OO) programs, its architecture allows for "global" functions, meaning that a completely procedural program can be analyzed as well. The interaction events being analyzed include messages (function invocation/return), and can be extended to object creation/destruction and data read/write.

This tool is intended to support design recovery, design/implementation validation, and reengineering type tasks. Some examples include the following:

1) An implemented system is to be examined by the implementers to validate its correspondence to the intended design. The system is instrumented and exercised with specific usage scenarios and trace files generated. Design level scenarios are entered into ISVis and compared with the implementation behavior. As the behavior is examined, areas of divergence and incorrect or non-optimal behavior become apparent. The implementation is reengineering based on these observations.
2) A legacy system is to be modified with extended functionality. Before this can be done, the architecture and design of the existing system must be established and understood. By

using the instrumented system in various scenarios, trace files are generated. In the trace scenarios repeated patterns of interactions are observed. As these patterns are examined and understood, higher-level scenarios are created. Eventually, design-level scenarios are created from the low-level interaction scenarios.

Features of ISVis include the following:

- analysis of program execution traces numbering over 1,000,000 interactions
- simultaneous analysis of multiple traces from the same program
- views include actor and interaction lists and relationships, scenarios, and source code (via XEmacs)
- use of Information Mural visualization techniques to portray global overviews of entire interaction scenarios
- abstraction of actors through containment hierarchies and user-defined components
- selective filtering of single or all occurrences of a particular interaction
- construction of higher-level scenarios using repeated sub-scenarios (interaction patterns)
- identification of scenarios used as patterns for location of the same or similar scenarios in executions
- user-specifiable scenarios, including regular expression wildcards for actors, used as patterns
- capabilities for saving and restoring analysis sessions

Impact

The ISVis tool has proven useful in assisting real-world software engineering tasks, and its full potential has not yet been realized. The next section describes two case studies in which ISVis is used to support design recovery and reengineering of several different systems, including the ISVis tool itself.

Case Studies

In order to validate the usefulness of the hypotheses embodied in this research, several case studies were performed using the tools to support program understanding tasks. As mentioned previously, this work is focused in particular on assisting design recovery, design/implementation validation, and reengineering tasks.

Case I: Reengineering View Redraw Mechanisms in GUI Applications

The first case arose because of the difficulty in correctly designing the drawing and layout mechanisms in X Windows applications. In response to a user manipulating windows on the screen, different expose and resize events are generated and forwarded to the appropriate applications currently executing. The application itself is responsible for redrawing the contents of so-called generic "drawing area" widgets, if necessary. Information visualization applications are increasingly being used with larger and larger data spaces, and redrawing the visual representations of large data sets can be expensive even with today's processor speeds. It is important that such visualization applications be designed to perform changes to the visual display as efficiently as possible.

This case is divided into two parts, using two different subject visualization applications built using X Windows and Motif. The first application is the ISVis tool developed as part of this research. The second is an algorithm animation built using the Polka toolkit. The purpose of this case is to see if the applications are correctly and efficiently managing their visual displays as users manipulate their windows, and if not to suggest possibilities for reengineering. Additionally, by examining two different systems which utilize the X Window system, the tester's hypotheses regarding its behavior can be validated.

Details

This section details the steps performed in the first case study and can be skipped if the reader wishes to read only the summary of results from the case. The first part of the study is performed by the author of the system itself. Thus, the person performing the analysis has (or thinks he has) an intimate knowledge of the subject system. Before beginning the use of ISVis to analyze itself, it was clear that the redraw and layout mechanism was not operating correctly. The focus area of the Scenario View is not drawn at all when the window first pops up, but the mural area is drawn. The designer's hypothesis about how the X Window system generated events is that uncovering part of a window on the screen leads to an expose event, resizing a window smaller leads to a resize, and resizing a window bigger leads to a resize and an expose. The ISVis drawing and layout system was architected based on this hypothesis.

The description below indicates the steps performed during part Ia of the case study. Labels on the left include pre-conditions, actions taken using the ISVis tool (marked "ISVis"), actions performed externally to ISVis (marked "external"), usage scenarios of the instrumented subject system, and conclusions or hypotheses formed.

Part I: Examining ISVis

pre-conditions:
- Scenario View focus area not drawn on window creation
- Scenario View mural drawn on creation

external:
- compile ISVis with Solaris compiler option to generate static information database

ISVis:
- run on ISVis executable
- do static analysis
- instrument all files related to views and X

external:
- recompile to generate instrumented ISVis

usage scenario 1:
- run instrumented ISVis
- read in bsort.short trace
- open Scenario View (nothing drawn in focus area)
- resize window smaller (nothing drawn in focus area)
- resize window larger (contents of focus area drawn)

ISVis:
- run on ISVis executable
- read trace from usage scenario 1
- notice high volume of `GetAppContext` messages, create new scenario w/o them called "isvis"
- notice recurring message pattern which is doing the mural redraw in the background, select it and create a scenario "redraw mural work proc"
- replace exact occurrences of "redraw mural work proc" in "isvis"
- create scenarios for expected behavior of ISVis:
  - "resize" = View::resizeCB, ScenarioView::layout (layout not done)
  - "resize+layout" = View::resizeCB, ScenarioView::layout, View::setDirty (layout done)
  - "resize+layout+expose" = View::resizeCB, ScenarioView::layout, View::setDirty, ScenarioView::draw
  - "expose" = View::exposeCB, ScenarioView::draw (draw not done)
  - "expose+draw" = View::exposeCB, ScenarioView::draw, View::isDirty (draw done)
- browse trace and find exact interleaved occurrences of above scenarios in "isvis", notice following sequence of messages:
  within ViewManager::openScenarioView:

ScenarioView::update, ScenarioView::draw (not done)
ScenarioView::show, View::resizeCB,
ScenarioView::layout (not done)

View::exposeCB, ScenarioView::draw (not done)

Mural::ExposeCB, Mural::Draw, Mural::WindowCreated,
Mural::Layout, Mural::SetDirty, Mural::Draw, Mural::IsDirty,
ScenarioView::muralRedrawNeededCB

"resize+layout"
"resize+layout"
"expose+draw"

conclusions:
- calls to ScenarioView::draw and ScenarioView::layout as a result of ViewManager::openScenarioView do nothing
- first View::exposeCB event when window is created does not draw contents
- first Mural::ExposeCB event does draw contents of mural
- user resizing window smaller results in a single "resize+layout" which does not draw contents of focus area
  - the mechanism in the design which "forces" an expose by calling ScenarioView::draw from ScenarioView::layout if the window has shrunk does not seem to be working
- user resizing window larger results in "resize+layout" followed by "expose+draw" in which the contents of the focus area are drawn

external:
- consult source code:
  - in Mural::Draw, if window buffer isn't created calls a routine which checks if the widget is realized and then calls Mural::Layout which creates buffer
  - in ScenarioView::draw, if buffer isn't created just returns
    - however, window should be valid if expose was generated, so should call layout then and create the buffer

conclusions:
- putting a call to ScenarioView::layout in the ScenarioView::draw function to be called if the buffer isn't created should solve the problem in which the focus area is not drawn as a result of the first View::exposeCB

external:
- made above change to code
- executed ISVis and got a core dump

ISVis:
- instrument modified code

external:
- recompile instrumented ISVis

usage scenario 1b:
- perform usage scenario 1 again

ISVis:
- read trace from usage scenario 1b
- ScenarioView::draw called, buffer has not been created so Scenario::layout is called, the layout is not done (buffer not created), and then draw continues without a buffer and core dumps

conclusions:
- need to check buffer again after ScenarioView::layout, like is done in Mural::Draw

external:
- make change to code
- execute and problem solved!! Scenario View focus area is drawn when window pops up

ISVis:
- instrument modified code

external: - recompile

usage scenario 1c:- perform usage scenario 1 again

ISVis: - read trace from usage scenario 1c
- find exact occurrence of "resize+layout+expose":
  "resize+layout+expose"
  "resize+layout"
  "expose+draw"
- browse and look for other problems:
  ScenarioView::layout called but not done in ScenarioView constructor
  ViewManager::updateViews called after constructor completes, why?

conclusions: - mechanism to "force" expose when window is shrunk is working properly
- no reason to layout Scenario View in constructor
- ViewManager::updateViews should not be called in ViewManager::openScenarioView, this remained from an earlier version in which the Program Model was modified in the Scenario View constructor

Part II: Examining Polka depth-first search animation

pre-conditions: - behavior seems correct

external: - compile Polka and dfs animation using Solaris compiler to generate static information database

ISVis: - run on dfs executable
- do static analysis
- instrument all files

external: - recompile to generate instrumented dfs

usage scenario 2: - run instrumented dfs with dfs1.in input file, press start, press pause, resize window smaller, resize window larger, cover and uncover window, resize window smaller, resize window smaller

ISVis: - run on dfs executable
- read trace from usage scenario 2
- create scenarios to find "pauseCB", "refreshCB", and "resizeCB",

sequence:	user action v1	user action v3
pauseCB	pause	pause
refreshCB, View::Refresh	??	smaller
resizeCB, View::Refresh	smaller	|
refreshCB, View::Refresh	|	larger
resizeCB, View::Refresh	larger	|
refreshCB, View::Refresh	|	|
refreshCB, View::Refresh	expose	expose
refreshCB, View::Refresh	|	smaller
resizeCB, View::Refresh	smaller	|
refreshCB, View::Refresh	|	smaller
resizeCB, View::Refresh	smaller	|

conclusions: - view is redrawn in response to both expose and resize events
- based on hypothesis of X event generation, group events as resulting from user actions as in column v1
  - why does resizing smaller sometimes lead to a resize and a refresh but

not for the last one?
- why are there 2 refreshes for the expose?

usage scenario 3: - run instrumented dfs with dfs1.in input file, uncover window, resize smaller vertically, resize smaller horizontally, resize smaller horizontally, resize smaller vertically

ISVis:
- run on dfs executable
- read trace from usage scenario 3
- create scenarios to find "pauseCB", "refreshCB", and "resizeCB",

sequence:	user action v2
pauseCB	pause
refreshCB, View::Refresh	expose
refreshCB, View::Refresh	smaller
resizeCB, View::Refresh	|
refreshCB, View::Refresh	smaller
resizeCB, View::Refresh	|
refreshCB, View::Refresh	smaller
resizeCB, View::Refresh	|
refreshCB, View::Refresh	smaller
resizeCB, View::Refresh	|

conclusions:
- according to user actions in usage scenario must group events as in column v2
- look back at usage scenario 2 and construct grouping v3
- both of these groupings imply that resizing a window smaller leads to an expose and a resize, and resizing a window larger leads to an expose, a resize, and an expose
- the Polka design redraws the entire window in response to both resize and expose events, so that if a window is resized larger the window is redrawn three times, and if a window is resized smaller the window is redrawn twice
  - could use a dirty flag and an offscreen buffer like in ISVis design to only redraw once and then just blit to refresh
  - when Polka designer told about the problem, says "Aha, I can just move the drawAll call into the resizeCB and take it out of View::Refresh because refresh blits the offscreen buffer already, as long as View::Refresh is not called anywhere else..."

ISVis:
- check that in fact the View::Refresh is only called by resizeCB and refreshCB for the two usage scenarios

Part III: Back to ISVis

ISVis:
- run on ISVis
- read in usage scenario 1c
- check that an expose always occurs before a resize, and that the resize of a window larger also triggers an expose afterwards: yes!!

conclusions:
- design of ISVis to "force" generation of an expose when ScenarioView::layout indicates window was resized smaller is correct
- hypothesis that a window resize smaller in X triggers an expose and a resize, and that a window resize larger triggers an expose, resize, expose is valid

Results

The first case study shows the effectiveness of the ISVis tool in assisting dynamic program understanding during design/implementation validation of ISVis, design recovery of Polka, and reengineering of both applications. Results can be summarized as follows:

- improper design of the ISVis Scenario View drawing mechanism was revealed and through comparison with Mural drawing mechanisms reengineered to function correctly
- several inefficiencies in the design of ISVis were eliminated by examining behavior
- the ISVis designer's mental model of X Window system events generated as a result of window manipulation was determined to be wrong and a new correct model was gleaned by examining the behavior of ISVis and Polka applications
- inefficient but correct drawing and layout mechanisms in the Polka application were revealed
- Polka designer was enlightened by new model of X event behavior and reengineered a solution to fix design inefficiency

Case II: Design Recovery of Mosaic WWW Browser

The second case study involves reverse engineering the behavior of the Mosaic WWW browser. This case study arose as part of a larger effort to support the evolution of legacy systems (the MORALE project). Given an existing system and a new set of requirements, the idea is to support the reengineering process by identifying what the system can do in its current state, what needs to change to support the new requirements, and then make suggestions as to how the system should be changed to accomplish the new mission. The first system being analyzed as part of this work is NCSA Mosaic version 2.4, with a new scenario of being able to handle user-configurable external viewers.

A part of the MORALE process involves establishing the current architecture of the subject system, as it relates to the particular change scenarios. This is a design recovery task, and might be called architectural localization. Without a correct model of the current system, the rest of the reengineering process cannot proceed with any accuracy. The ISVis tool can help an analyst construct an accurate model of the system's behavior in given scenarios, and help validate hypothesized models of the system.

Details

The basic strategy for using ISVis in this design recovery task is the following:

1) compile Mosaic using Solaris CC compiler to generate static information
2) use ISVis to read the static information and generate instrumented source code
3) compile the instrumented Mosaic
4) generate execution traces by exercising Mosaic in related usage scenarios, such as following a hypertext link, displaying a PostScript file, and popping up internal Mosaic windows
5) read the execution traces into ISVis
6) create working scenarios and build up to design-level models using the following tactics:
   a) abstract the view of scenarios by using the natural containment hierarchy
   b) eliminate leaf messages and others which appear unrelated to the specific change scenario
   c) use the global overview or ask ISVis to identify interaction patterns
   d) understand interaction pattern behavior, and replace the low-level interactions with a reference to the recovered scenario
   e) use pattern matching to locate similar scenarios
   f) investigate the behavior of actors by viewing their source code
   g) build components of related actors which seem to provide related, cohesive functionality
7) view the resulting design-level components and scenarios and store analysis results

usage scenario 1: - run instrumented Mosaic, starts at CoC home page, follow People link, follow PhD link, scroll to Dean Jerding, follow Dean Jerding link,

follow resume (local PS file) link (opens ghostview), follow paper ("file:" PS) link (opens ghostview), follow resume (local PS file) link (opens ghostview)
- generates m3.trace

ISVis:
- start ISVis on Mosaic 2.4 executable
- read in m3.trace
- create m3 working scenario as a copy of m3.trace
- create components for the libXmx, libhtmlw, libwww3 subdirectories
- view scenario with those components and all files in Mosaic/src
- browse through scenario, removing messages which seem unrelated (color them magenta)
- notice "image resolve" scenario for images being processed, by finding occurrences of ImageResolve at recurring patterns in the global mural
- mo_open_window done once during start up
- view source of HTSACopy, it is a string management, done a lot, remove
- notice "HTMIME put character" and "anchor visited predicate" scenarios
- url_canonicalize done in mo_www.c
- globalhist.c seems to cache data, as in the "cache image" scenario--this is done after each "image resolve"
- find an anchor_cb, then find it six times--this is the click on anchor in Mosaic callback
- mo_set_text starts parsing page of html text
- mo_load_window_text is done for all anchor_cb's, as is mo_pull_er_over--created scenario for each "anchor_cb"
  - difference is there is an HTSimplify in the anchor_cb for the "file:" PS
- for html pages, after return from mo_pull_er_over, mo_do_window_text is called
- for PS, mo_here_we_are_son, mo_redisplay_window, mo_done_with_icon, and mo_not_busy
- looking for the decision context to display ghostview:
  - seems to be mo_load_window_text? view the source in gui_documents.c
  - a flag use_this_url_instead indicates to mo_here_we_are_son
  - is path in line 720 taken? is new text = <mosaic-access-override>?
    - view mo_pull_er_over, do_it, and _HTMLInput
      - aha, htmlw is the html widget--saw that in HTML.c
  - where is the damn ghostview call?
  - look at HTLoadAbsolute source, in HTLoad there is an indirect function call to load the document depending on the protocol
    - HTLoadFile called once for the "file:" protocol, all the rest use HTLoadHTTP
      - find HTLoadExtensionsConfigFile (extensions for filenames)
    - find HTLoadTypesConfigFile, ProcessMailcapFile, and HTSetPresentation in the beginning of the trace
      - storing presentation info for various formats, including ghostview for PS
  - now using the primary and related focus, finding callers of HTListObjectAt, which is HTFilesSuffix, which is called by HTSaveAndExecute
    - this sounds interesting, view source
      - BINGO, this is where the decision is made to invoke an external viewer on a temp file
      - the actual call is made in HTLoad... when the pointer in the

end_document of the presentation structure is looked at
- HTSaveAndExecute is called within HTFTPLoad (file: PS) and HTLoadHTTP for rest

Elapsed time: 4 hours--save session m3.session5

ISVis:
- restore session m3.session5
- now try and abstract up to design level, make “m3 design” scenario as copy of “m3 working”
- remove Xmx.c, Xmx2.c -> core dump (fixed bug in InteractionImpl::removeActor)
- don’t do that, because will remove scenarios involving those actors...
- browse “m3 design” scenario again, removing unrelated interactions (color blue) and use find pattern to locate repeating scenarios
- get_physical is where the protocol for the anchor is set
- HTAccess keeps list of protocols
- HTFormatInit initializes format conversions
- what does StreamStack do?
- fire_er_up is done once at beginning to create the window for the local home page
- the previously created scenario “get physical image” is not done just for images
- expand scenario causes core dump...
- processing of presentation list done all over
- identify several “image resolve 2” scenarios, which differ only in calling context
- save session m3.session9
- during the processing of the third link followed in the scenario, there is an “image resolve” scenario that causes HTFileInit to load suffixes
- found an anchor_cb ‘ scenario for other PS files opened
  - in first, HTSaveAndExecute calls HTFileSuffix
  - the second PS file is brought over using HTLoadFile (it is a link with a “file:” reference)
    - this is a “get physical file” scenario
  - “get physical image” scenario should be “get physical http”
  - get_physical is where the protocol decision is made as to what to load

Elapsed time: 7 hours--save session m3.session10

usage scenario 2:
- run instrumented Mosaic, starts at CoC home page, open hotlist, scroll, dismiss, open window history, dismiss
  - generate m4.trace

ISVis:
- read in m4.trace and save a copy as “m4 working”
- find return from fire_er_up
- found a menubar_cb, then a mo_post_hotlist_window
  - this is in the hotlist file, seems that putting up the hotlist window is the responsibility of the hotlist component
  - mo_load_hotlist_list to fill window, using the default_hotlist data structure
- found another menubar_cb, then a mo_post_history_win
  - responsibility of the history component (not global history)
  - mo_load_history_list to fill window
  - each Mosaic window data structure keeps its own history, in addition

to the global history

Elapsed time: 7.5 hours--save session m34.session

ISVis:
- now using all the information and scenarios defined thus far, try and group functions and files into components
  - analyst is also guided by the proposed SAAM architecture
- history related components:
  - global (globalhist.c)
  - window (history.c)
  - cache (globalhist.c)
- "par" is personal annotations
- libhtmlw/HTML.c is the html widget
- mo_www.c manages loading of documents
- libwww2/HTAccess.c is Access Manager
- libwww2/HTAlert.c lets Mosaic know progress of loads
- libwww2/HTInit.c initializes formats with HTFormat.c
  - the latter keeps track of MIME types and how to display them
  - these are the Presentation Manager
- libwww2/HTMIME.c is stream/message parsing for html, HTTP
  - goes in Stream component
- Protocol Manager
  - libwww2/HTTP.c speaks http
  - HTFile.c, HTFTP.c also
  - HTTCP.c used by all, it goes in Stream component
- gui.c and gui_documents.c and gui_menubar.c are UI related

Elapsed time: 8.25 hours--save session m34.session2

- HTFormat.c involved in reading FTP using HTGetCharacter
  - Stream component also includes some HTFormat functions
- HTMosaicHTML sends characters over the Mosaic (mo_www.c)
- browse through remaining unassigned files and functions, look at comments, and determine possible component mapping
- end up with 15 components (1st 15 in list)
- note that end_document is called in HTLoadHTTP and HTFTPLoad
  - this is where ghostview is executed based on what is stored in HTSaveAndExecute

Elapsed time: 9 hours--save session m34.session3

Results

During the second case study, the ISVis tool assisted with the reverse engineering of a legacy system's behavior in particular usage scenarios. Over a period of nine hours, an analyst unfamiliar with the system being examined and somewhat familiar with the domain was able to construct a component architecture grouping source code entities by the role they play in the particular usage scenarios. Almost fifty interaction patterns were identified and understood during the session. The analyst identified the scenarios in two ways: 1) the global overviews provided initial insight to similar locations in the traces and by navigating to these locations similar sequences were observed, selected, and defined as scenarios, and 2) the automatic detection capabilities located repeated message sequences, which if appropriate, were selected and defined as scenarios. When questions arose during the examination of traces, being able to open views of the source code proved indispensable for understanding in more detail the role of specific functions and data. The

component grouping facilities allowed the analyst to map the functions and files in the system into 15 separate components which seemed to divide the functionality of the subject system in the particular usage scenarios. Creating the interaction scenarios and building a "higher-level" view of the execution traces allowed this behavioral understanding task to be accomplished; without the ISVis tool (using only textual traces, a debugger, and source code) an understanding of the system would have required many times as much effort, if possible to achieve at all.

Conclusion

The techniques and methods embodied in the tools built during the course of this research have been tested in several case studies, revealing the usefulness and effectiveness of these tools. The results of the these studies, along with the published research accomplishments, have validated the thesis of this work, namely that visualizing interaction patterns in program executions can facilitate program behavioral understanding during design recovery, design/implementation validation, and reengineering tasks.

References

[1] De Pauw, Wim, Richard Helm, Doug Kimelman, and John Vlissides. *Visualizing the Behavior of Object-Oriented Systems*. Proceedings of the ACM OOPSLA '93 Conference, 1993, pp. 326-337.

[2] De Pauw, Wim, Doug Kimelman, and John Vlissides. *Modeling Object-Oriented Program Execution*. Proceedings of the European Conference on Object-Oriented Programming, 1994.

[3] Eick, Stephen G., Joseph L. Steffen, and Eric E.Sumner Jr. *SeeSoft: A Tool for Visualizing Line Oriented Software Statistics.* IEEE Transactions on Software Engineering, vol. 18, no. 11, 1992, pp. 957-968.

[4] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Pattterns: Elements of Reusable Object-Oriented Software.* Addison -Wesley, 1995.

[5] Goldberg, Adele. *Smalltalk-80, The Interactive Programming Environmen.* Reading, Pa: Addison-Wesley, 1983.

[6] Jerding, Dean F. and John T. Stasko. *The Information Mural: A Technique for Displaying and Navigating Large Information Spaces.* Proceedings of the IEEE Visualization `95 Symposium on Information Visualization, 1995, pp. 43-50.

[7] Stasko, John T. and Eileen Kraemer. *A Methodology for Building Application-Specific Visualizations of Parallel Programs.* Journal of Parallel and Distributed Computing, vol. 18, no. 2, 1993, pp. 258-264.