Using Visualization for Architectural Localization and Extraction

Dean Jerding and Spencer Rugaber
College of Computing
Georgia Institute of Technology
{dfj,spencer}@cc.gatech.edu

Abstract

Understanding the architecture of a program requires determining both the major components into which the system is broken and the ways in which the components interact to accomplish the program's goals. Both static and dynamic analyses of the software can aid in obtaining this understanding. This paper describes an analysis technique for gaining such understanding and a visualization tool, called ISVis, that supports it. The technique is applied to the problem of enhancing the Mosaic web browser by both visualizing its architecture and finding the components of the browser into which an enhancement should be inserted.

Keywords: software architecture extraction, program visualization, dynamic analysis, program understanding

1. Understanding Software Architectures

Software development typically means software enhancement. And to successfully enhance software first requires understanding it. There are many aspects of program understanding, but one of the most essential is understanding a program's architecture—its major components and their interactions. In particular, inserting an enhancement at the optimal location in a complex program can improve the long-term maintainability and reusability of the program. This paper describes a technique for understanding programs at the architectural level with emphasis on finding locations in a program where making enhancements is particularly appropriate.

A *software architecture* is a high-level program model that describes a system's major pieces (its *components*) and how they interact (its *connectors*). Such models are often graphically depicted with boxes denoting components and arrows indicating connectors. Extracting an architectural model from a program traditionally means treating subprograms as components and subprogram invocations as connectors. Extraction at this level is straightforward but may generate too much detail. Among the few successes that go further are Harris et al. [8] and Fiutem et al. [5], both of whom rely on lexical cues to detect specific system-library invocations. For example, Harris et al. equates components with Unix processes and specifically looks for instances of the term *fork* to identify them.

Program analysis to support software understanding takes one of two forms: *static,* where the program code itself is analyzed, and *dynamic,* where the program is executed to learn how it behaves. Although static analysis is commonly practiced and entirely appropriate for determining structural properties such as architectural components, dynamic analysis is a better match for determining behavioral properties such as component interactions.

This paper describes a program understanding technique that combines static and dynamic analyses to extract components and connectors. The process has been used both to obtain an overall view of a system's architecture and to solve the specific problem of where in the system to insert an enhancement. We call the latter problem *architectural localization.*

ISVis (Interaction Scenario Visualizer) is a tool that supports the process. It includes several graphical views with which an analyst can determine appropriate components and connectors. For ISVis, a component consists of any analyst-specified collection of underlying source code constructs. Component specification is supported by traditional static analyses. Connectors consist of component interactions as recognized from actual execution traces, a form of dynamic analysis.

The paper also describes the application of ISVis to a specific program, the Mosaic web browser, version 2.4. The task we studied was the addition of user-configurable viewers to Mosaic. Version 2.4 supported viewers for particular kinds of data, such as PostScript, but the user could not dynamically add new ones. We wanted to know enough about the architecture of Mosaic to determine where in the source code to add the new capability.

2. Approach

This section describes the approach we have taken to extracting software architectures. Two key aspects of the approach are visualization and abstraction.

2.1 Visualization

The use of graphical techniques to depict information on a computer display has proven to be useful for analyzing various forms of information, including computer programs. Visual representations of the voluminous information that can be derived from program executions are a powerful means for that information to be processed and analyzed. While off-line analysis of the data is useful in its own right, we believe that visualizations supplementing the human pattern-recognition and abstraction capabilities better support such a complex process. Our approach is to provide an analyst with a process and a tool within which a program's behavior can be visualized, filtered, and abstracted and with which the analyst can build and save views of the behavior appropriate for the particular program understanding task.

2.2 Abstraction

Through the use of visualization prototypes built during the course of this research [10], it has been observed that program executions are made up of recurring patterns of interaction, manifested as repeated sequences of program events such as function calls, object creation, and task initiation. Instances of these interaction patterns occur at various levels of abstraction. Using them, the analyst can help bridge the gulf of abstraction between low-level execution events and high-level models of program behavior. Humans typically solve complex problems by using divide-and-conquer strategies, by detecting patterns, and by finding analogies; interaction patterns can be used in all three of these activities.

2.3 Terminology

This section describes the conceptual basis of ISVis in terms of a sequence of definitions. Definitions include a description and a (possibly empty) set of attributes. In addition, each defined term has name and description attributes that provide, respectively, a unique identifiers and arbitrary, analyst-supplied descriptive text for each instance of the items being defined.

Architectural components denote specific source-code constructs. To extract an architecture means detecting a meaningful abstraction and the associated source code that implements the abstraction. We call the source code units *actors.* Actors can be *simple* (mapping directly to code) or *composite* (made up of lower level actors). A component is an abstraction and the corresponding actors that constitute it.

> **Definition** a *simple actor* is a syntactically identifiable program unit; for example, a function, an object, or a data item. As the last possibility suggests, actors can be passive as well as having computational capabilities. An actor has a **location** (the position of the actor's definition in the source code) and a **type** (the actor's syntactic type).

> **Definition** a *composite actor* is a set of actors each of which is either simple or itself composite.

> **Definition** an *actor* is either a simple or a composite actor.

> **Definition** a *component* is an actor and the corresponding abstract **role** that it serves in the architecture.

The job of the analyst is to locate and define components in the source code by composing actors or previously recognized components and then determining the roles the components fill.

Similarly, the analyst is responsible for detecting connectors. In this case, the low level unit is an event that takes place during program execution.

> **Definition** an *event* is a discernible unit of program execution. These can be generic, like the invocation of a function or method, function return, object creation or deletion, or data reference; or they can be specified by the analyst, indicating specific execution events that the analyst wishes to track. An event has a **time stamp** (a record of when the event took place). Time stamps provide a serial ordering to events. An event also has a **type** (an analyst-specified identifier enabling similar events to be associated).

> **Definition** an *event trace* is a record of the events that occur during an execution of a program.

Events take place in the context of one or more actors. For example, an event might be the invocation of one subprogram actor by another. The combination of an event with its associated actors is called an *interaction.*

> **Definition** an *interaction* is a relation between an event and one or more actors.

Sequences of interactions then form *interaction scenarios*. Interaction scenarios can be generated from executing an instrumented program with particular input data, or might be specified directly based on design models.

> **Definition** an *interaction scenario* (or simply a *scenario*) is a sequence of interactions. Note that the sequence obeys the time ordering of its constituent events, but that the events in the sequence are not necessarily contiguous with respect to the underlying event trace.

> **Definition** a *usage scenario* is the execution of a subject program with a given set of input test data. A usage scenario leads to the generation of multiple interaction scenarios.

The connector abstraction process that the analyst uses with ISVis is one of pattern detection. The analyst uses ISVis' visualization features to detect recurring interaction scenarios. If a recurring sequence of interactions is meaningful to the analyst, it can be characterized by an interaction pattern*,* and the analyst can use ISVis to replace instances of the pattern with a higher-level abstraction in the displayed visualization.

> **Definition** an *interaction pattern* is a description of recurring scenarios. The pattern specifies an ordered list of interactions, where one or more of the elements of the list can be a wildcard denoting interpolated

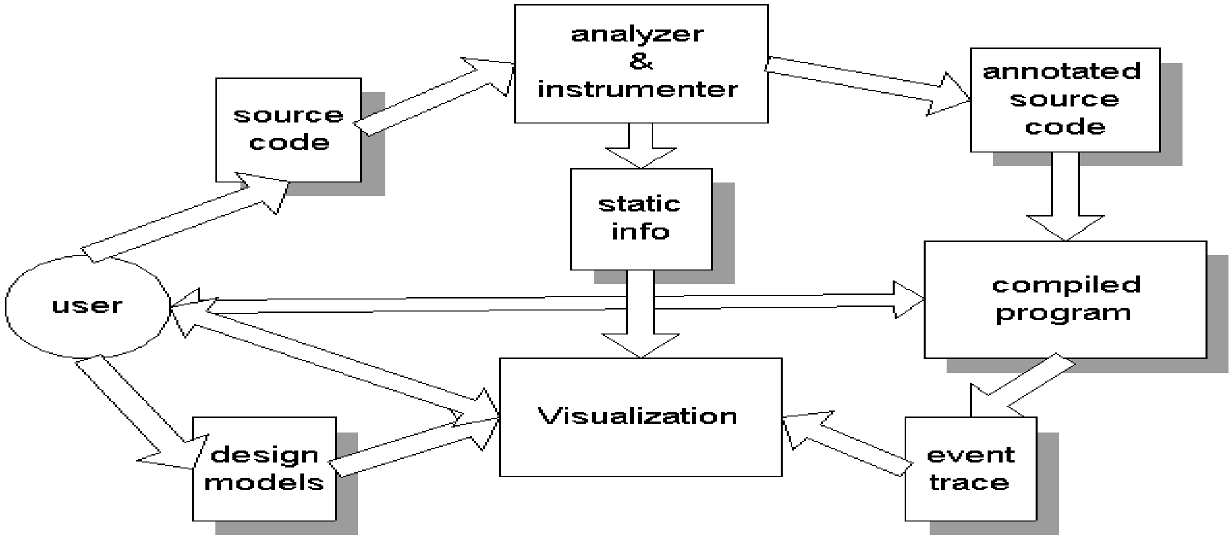

Figure 1: Process Overview

events that do not conceptually contribute to matching scenarios.

It is possible to classify instances of any of these definitions. In particular, actors and events have primitive type information as part of their attributes. Components, interactions, and scenarios are composed from actors and events and, consequently, derive from them a unique type signature. It is possible to think, therefore, of the class of all actors of integer type. Another example is the class of all scenarios involving two events, the first of type *call* and the second of type *return* where the four associated actors (*caller*, *callee*, *returner*, and *returnee*) have identical **locations**. We might call this latter class of scenarios *recursive invocations*. Class definitions form a concept hierarchy enabling an analyst to overlay abstractions on the underlying events and actors.

2.4 Process Overview

The overall process of performing architectural localization is depicted in Figure 1. It comprises a static analysis of the subject system, instrumentation of that system to track interesting events, execution of the instrumented system in particular usage scenarios to generate event traces, and visualization and abstraction of the event traces using the ISVis tool. Design models can also be entered into ISVis in the form of interaction diagrams to compare with actual system behavior.

3. ISVis

The purpose of ISVis is to support the browsing and analysis of event traces derived from program executions. It is useful during software engineering tasks requiring a behavioral understanding of programs, such as design recovery, architecture localization, design or implementation validation, and reengineering. Features of ISVis include the following.

- analysis of program event traces numbering over 1,000,000 events
- simultaneous analysis of multiple traces for the same program
- views include actor and interaction lists and relationships, scenarios, and source code (via XEmacs)
- use of Information Mural [9] visualization techniques to portray global overviews of scenarios
- abstraction of actors through containment hierarchies and analyst-defined components
- selective filtering of individual or multiple occurrences of a particular interaction
- definition of higher-level scenarios comprising repeated sub-scenarios
- identification of scenarios to be used as patterns for locating the same or similar scenarios in other scenarios
- analyst-specified interaction patterns, including regular expression wildcards for actors
- saving and restoring of analysis sessions

3.1 Architecture

Figure 2 shows an architectural diagram of ISVis, including its components, connectors, and input–output files. The Static Analyzer reads the Source Browser database files produced by Solaris compilers and generates a static information file. The Instrumentor takes the source code, the static information file, and information supplied by the analyst about what actors to instrument (specified in the trace information file), and generates instrumented source code. This source must be compiled externally to the ISVis tool, and then, when the instrumented system is executed using relevant test data, event traces are generated. The Trace Analyzer in ISVis uses the trace information files and the Event Stream to read event traces and convert them into scenarios, stored in the Program Model. As scenarios are created, the actors involved are also

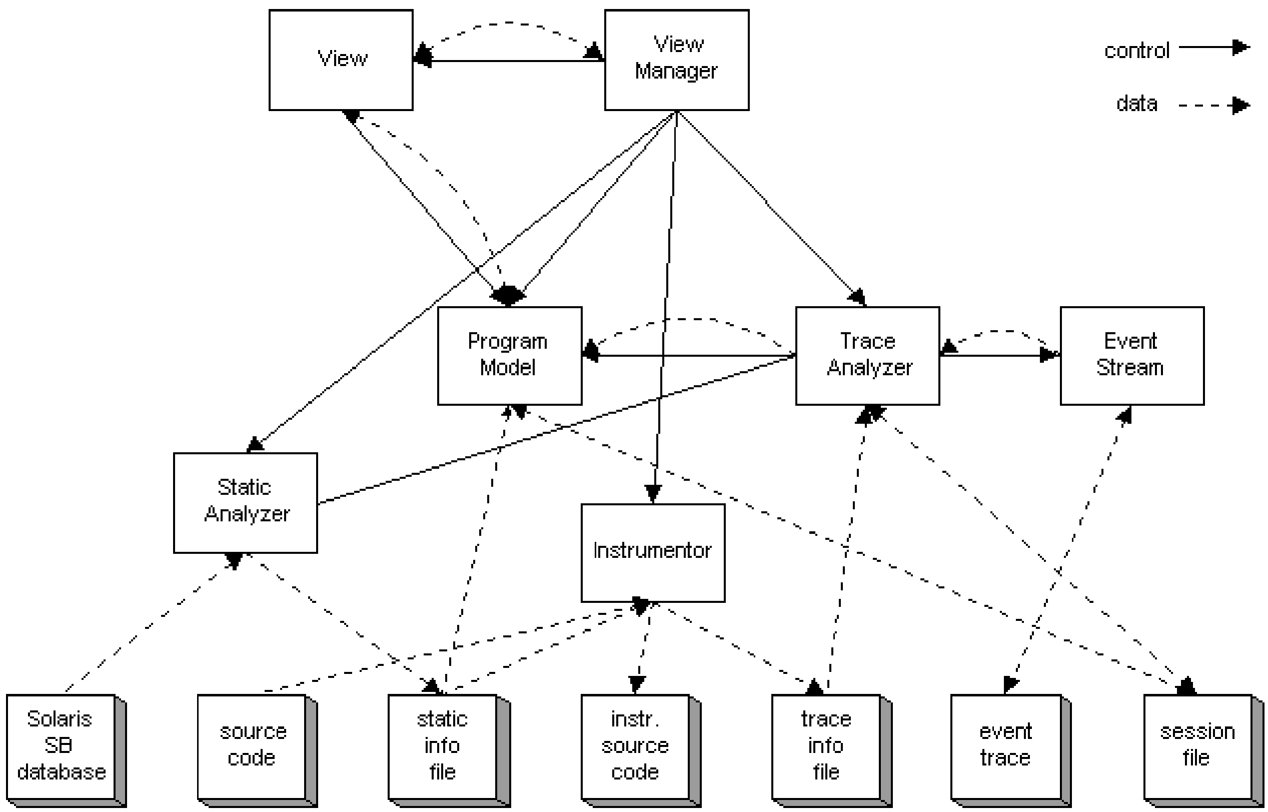

Figure 2: ISVis Architecture

added to the Program Model. The user then interacts with the Views of the Program Model to do the analysis. A Program Model can be stored for later use in a session file.

3.2 Example Views

ISVis currently provides two views, the Main View and the Scenario View. Figure 3 is a snapshot of the Main View during the case study described in Section 4. The top portion of the view lists the actors in the Program Model, including user-defined components, files, classes, and functions. The middle portion includes lists of the scenarios and interactions in the Program Model, as well as an area for displaying information about the item in primary focus (selectable with the middle mouse button). The key area allows users to assign colors to actors or interactions that have been selected using the left mouse button. The bottom portion of the view is a shell for textual information input–output. Note that each of the scrollable lists of actors and interactions uses an Information Mural [9] to display a graphical overview of the selected and colored items in the list.

The Main View includes a menu bar for entering commands, including the ability to open a Scenario View for each scenario in the model. Figure 4 shows a Scenario View from the case study described in Section 4. The Scenario View is in fact a Temporal Message Flow Diagram (TMFD) [2], sometimes called an interaction diagram, message sequence chart, or event-trace diagram. Actors in the view are assigned columns, and interactions are drawn as lines from source to destination actor in descending time order. A global overview of the scenario appears on the right of the view, and is used to navigate through the interactions in the scenario. The overview is created using an Information Mural, which provides effective global overviews of scenarios containing hundreds of thousands of interactions. The Mural uses techniques much like anti-aliasing to preserve visual characteristics as if the analyst could see the entire Scenario View from a distance. This allows the analyst to observe various phases in the scenario including repetitive visual patterns, indicating the presence of interaction patterns in the subject program execution being analyzed. As interactions are selected and colored the Mural is colored as well, helping an analyst locate where particular interactions occur in a program's execution.

The Scenario View provides several features to help an analyst build abstract models of the subject system and to localize behavior. An option menu allows the actors in the scenario to be grouped by containing file, class, or component actors. Another option allows the user to select a class of interactions or just a single instance of an interaction. Once a sequence of interactions are selected, they can be defined as a scenario (added to the Program Model) and then all occurrences of that sequence of interactions in the original scenario are replaced with a reference to the

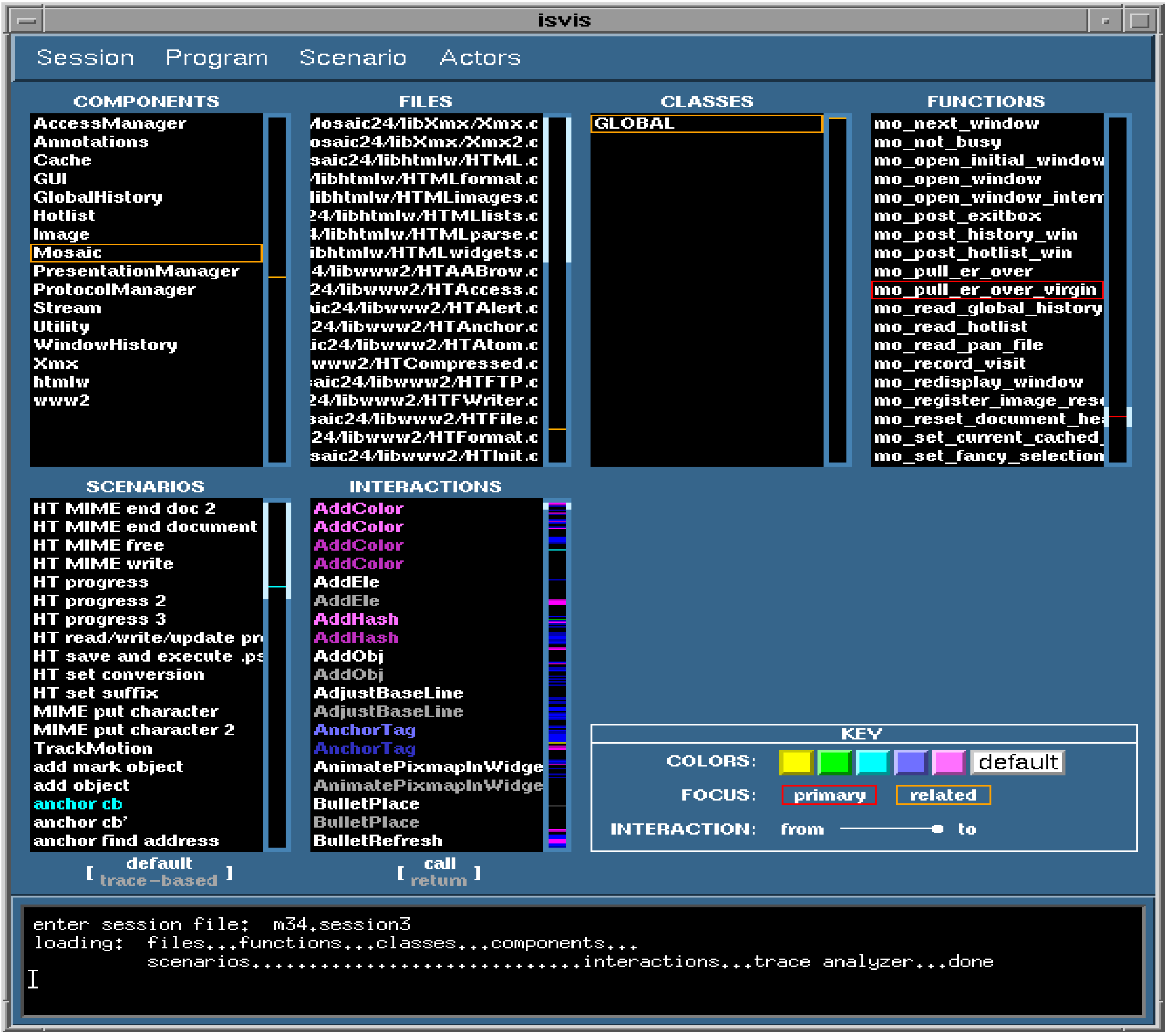

Figure 3: ISVis Main View

newly defined scenario. While a simple interaction is a shown as a line connecting the source and destination actors, a sub-scenario that occurs within the Scenario View appears graphically as a rectangle containing all of the actors involved in the scenario.

The Scenario View also includes features to find interaction patterns in a scenario, in a manner similar to regular expression matching. For example, given an interaction pattern, the user can choose to look for an exact match in the scenario (actors and interactions match exactly), an interleaved match (all interactions in the pattern occur exactly, but others may be interleaved), a contained exact match (actors in the scenario contain the actors in the pattern, and the interactions occur in exact order), and a contained interleaved match. Additionally, actors in an interaction pattern may be specified with wildcards, meaning they match any actor. The last of the pattern features includes the ability to ask ISVis to look for repeated sequences of interactions that occur in the scenario. This helps an analyst locate sequences of interactions which may have a higher-level meaning in the system, in addition to the analyst simply noticing these patterns in the global overview or as he or she browses through the scenario.

Note that ISVis' two Views have a Subject-View relationship with the Program Model such that any selection or modification done in one view is immediately reflected in the other. Also, it is possible to save the current Program Model and event traces that have been read in for later analysis.

4. Architectural Localization Case Study

Our case study arose as part of a larger effort to support the evolution of legacy systems (the MORALE

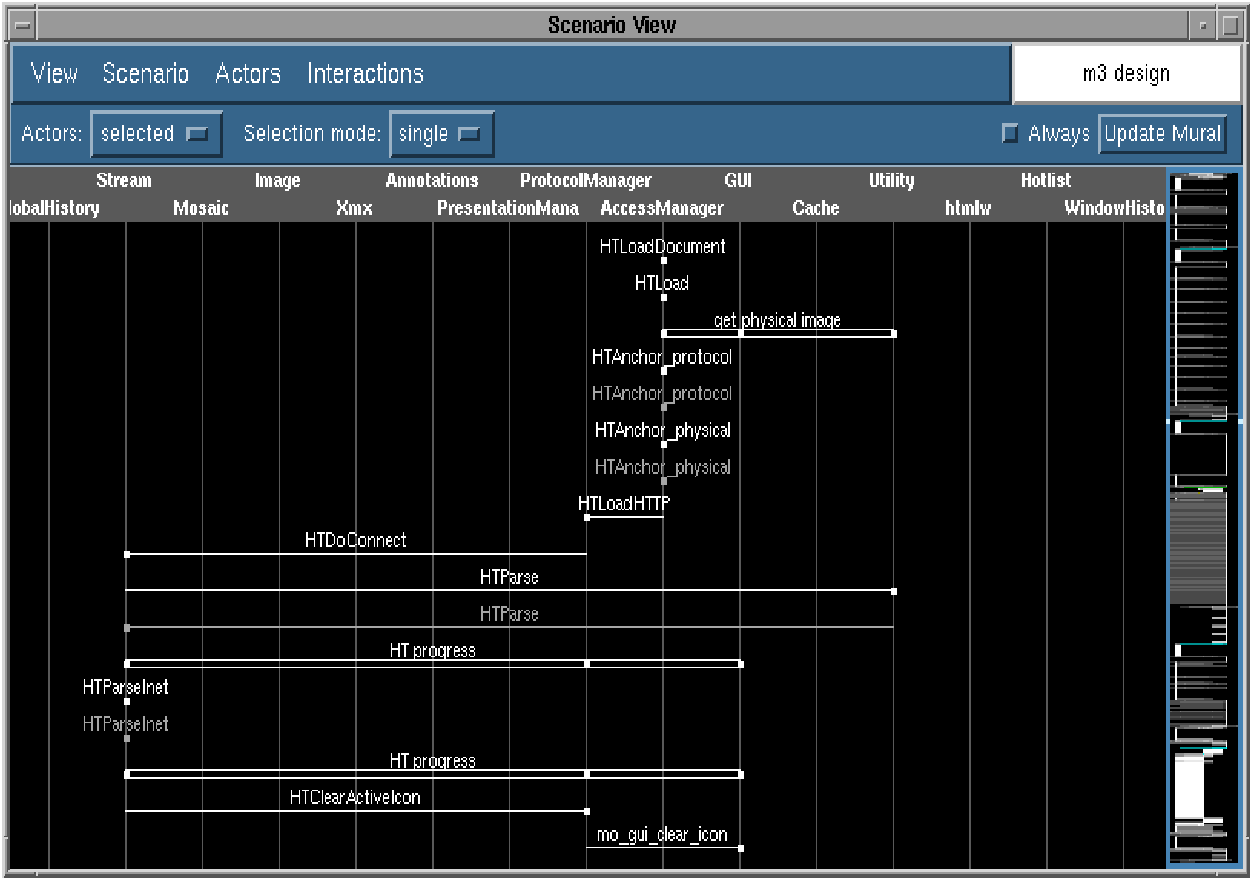

Figure 4: ISVis Scenario View

project) [1]. Given an existing system and a new set of requirements, the MORALE process identifies what the current version of the system can do and what needs to change in order to support the new requirements. It then suggests how the system should be changed to accomplish its new mission.

Part of the MORALE process ascertains the current architecture of the subject system as it relates to particular changes in requirements. This is an architectural localization task. Without a correct architectural model of the current system, the reengineering process cannot proceed with any accuracy. The ISVis tool helps the analyst construct an accurate model of the system's behavior and validate hypothesized models of the system.

4.1 Adding configurable viewers to Mosaic

The subject system for this case study is the NCSA Mosaic web browser, version 2.4 [15]. This version uses MIME [16] types to denote internal and external viewers for different types of web pages. The enhancement task is the extension of version 2.4 to support user-configurable external viewers, whereby Mosaic provides users interactive control over which viewers are used for specific types of web pages. The first steps in the reengineering process are understanding which parts of the system implement relevant functionality and which components must be changed or added to support the new requirements.

4.2 Process description

Initially, we had no understanding of the Mosaic implementation, and it was too large (100,00 lines of code) for a comprehensive study. We wanted to find out how Mosaic dealt with viewer invocation and where in the code this was done. We hoped that a simple generalization to this part of the code, we could avoid a complete source code analysis. Hence, we wanted to perform an architectural localization.

Architectural Localization: The process for using ISVis in an architectural localization task can be summarized by the following steps.

1. Compile the subject system using a Solaris compiler to produce static information. The Solaris compiler generates a Source Browser database that contains static information such as the program's symbol table, line-by-line scope, and call graph.
2. Use ISVis to read the static information and generate instrumented source code. ISVis uses PERL scripts to translate the native Solaris source browser database files into an ASCII static information file consumable by ISVis. It then provides an interface to instrument

the files, classes, and functions in the subject system. Another script places tracing objects in the code based on an event trace information file.

3. Compile the instrumented system. The Solaris C++ compiler is used because its tracing library provides objects to track function invocations.
4. Generate event traces by exercising the subject system in relevant usage scenarios. This is an important step in the analysis process, during which the analyst must determine which usage scenarios exercise behavior in the subject program related to the functionality that needs to be understood. This means uncovering those aspects of Mosaic that provide functionality related to how Mosaic determines which viewer to use for particular types of web pages and how Mosaic implements other user-controllable configurations. The following were the specific usage scenarios: following a hypertext link to an HTML file, following a link to and displaying a PostScript file, and popping up internal Mosaic windows with customizable settings. The event traces we generated consisted of almost 600,000 events.
5. Read the event traces into ISVis. ISVis reads event trace files and creates an internal model of the execution, including the actors and interactions involved in the scenarios.
6. Create working scenarios and build up architectural models.
7. View the resulting design-level components and scenarios, store analysis results, and iterate steps 5-7 as necessary. ISVis aids program understanding through an iterative process often requiring several analysis sessions. In this case study there were five separate analysis sessions, each building on the previous, over as many working days. The total time spent on the analysis was nine hours.

Architectural Localization Tactics: During the use of ISVis for this and other architectural localization tasks, we have identified a number of tactics useful for solving this class of problems. Some of these tactics are appropriate for solving other program understanding problems as well.

1. *Abstract the view of scenarios by using the natural actor containment hierarchy.* One of the most useful features of ISVis is the ability to project the interactions across the containment hierarchy of actors, including files, classes, and user-defined components. The scenarios from Mosaic include interactions between thousands of function actors, making viewing and understanding a scenario difficult at best. The first step in viewing the traces of Mosaic was to group actors by file. Next, files in particular subdirectories, such as the Xmx widget library, were further grouped into a single component because the analyst was not interested in the internal interactions between actors in those files, only in the interface between that component and the rest of the system.
2. Eliminate interactions unrelated to the functionality we are trying to localize. This capability allows an analyst to quickly locate, select, and removed unrelated interactions from scenarios. For example, we noticed and removed low-level string manipulation and graphics library calls completely unrelated to the task at hand. In the process of doing this, we also found other "utility" operations such as list manipulation, and grouped those actors together into a "Utility" component.
3. *Use the global overview and browse the scenario to identify interaction patterns.* The global scenario overview indicates phases in the scenario and also highlights areas of recurring sequences of interaction. It is thus possible to visually locate candidate interaction patterns by using the global overview to navigate to regions in the scenario where similar sequences of interaction occur. In the mural at the right side of Figure 4, you can see four different phases in the first two-thirds of the scenario, one for each HTML document visited. Repetitive patterns occur as each document is processed. Differences arise from the number of images in each document, another pattern that we found. Early on the analyst also discovered interaction patterns for the processing of a mouse click on an anchor, of which there are six in the scenario—three in the first two-thirds of the scenario for HTML links and three at the end for the PostScript documents displayed.
4. *Understand interaction pattern behavior and replace the low-level interactions with a reference to the recovered scenario.* Once a sequence of interactions has been identified as a candidate interaction pattern, the analyst should attempt to understand what that sequence of interactions does. If the interaction pattern represents an important, recurring task in the program, identify those interactions as a new scenario, add a scenario **description** to the model, and replace all instances of that set of interactions with a reference to the newly defined and understood scenario. This is how low-level events are abstracted up into design level behavior. Using this tactic, the analyst was able to reduce the number of interactions in the longest Mosaic event trace (450,000 events) by a factor of ten.
5. *Use pattern matching to locate similar scenarios.* In addition to visually locating an interaction pattern, ISVis provides simple pattern matching functionality to help an analyst find recurring sequences of interactions. ISVis can look for arbitrary sequences of interactions or for sequences that begin with a call interaction and end with the corresponding return interaction.
6. *Investigate the behavior of actors by viewing their source code.* Sometimes the analyst finds that different but closely related scenarios occur at various points in the execution of a program. ISVis allows the analyst to open views of the source code to actually look into a function and understand why particular interactions occur at some points but not others. During the latter part of this case study, the analyst began to open XEmacs views of the source code for various actors. When the analyst located the interaction "HTSaveAndExecute" in those interactions occurring after the handling of a PostScript link, it was confirmed by viewing the source code that this function was in fact where the external viewer for the link was determined.
7. *Build components out of actors that provide related, cohesive functionality.* Based on the understanding of

the system gained through browsing scenarios, identifying recurring interaction patterns, and viewing source code, an analyst can begin to group related actors into components. This furthers the abstraction of the low-level behavior up to the architectural level. The analyst used information gained by browsing the Mosaic scenarios, static information about actors such as the name of the file in which they are defined and their names in the program, as well as comments in the source code itself to help group related actors into components. For this case study the following components were identified: AccessManager, Annotations, Cache, GlobalHistory, GUI, Hotlist, Image, Mosaic (main control code), PresentationManager, ProtocolManager, Stream, Utility, WindowHistory, Xmx, and HtmlWidget. It should be noted that while sometimes all actors in a particular file seemed to fit nicely into a component, often actors in the same source code file were assigned to different components.

4.3 Results

During this case study, ISVis assisted with the understanding 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 an architecture of 15 components 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. These scenarios included the method by which presentation formats were read via MIME type specification and the method in which Mosaic receives a mouse click, finds the anchor, parses the URL and selects an external viewer if needed to display the contents of a URL. Additionally, architectural aspects of Mosaic such as the global or window history that can be viewed in pop-up windows by the user were understood to all combine the user-interface code with the underlying data code. The group of actors categorized as the PresentationManager component is where the functionality to configure the external viewers must be added.

4.4 Strengths and weaknesses

ISVis provided the analyst a means for building an abstract model of Mosaic's behavior as it executes particular usage scenarios. Its visualizations of the voluminous event trace information provided a framework within which the analyst can use human cognitive skills to make many of the abstraction decisions. An analyst can take advantage of application-domain and programming knowledge, as well as the source code itself, to make inferences that a completely automated tool could not perform well. By supporting the abstraction process via interaction patterns, ISVis performs the more compute-intensive processes such as replacing identified patterns and lets the analyst make the identification of which patterns are semantically important, a task-dependent decision.

Because ISVis is such a powerful tool, we often find ourselves coming up with new features that might be useful. One of the obvious ones is to take advantage of off-line pattern-matching computation to more effectively suggest patterns the analyst. Another feature is the ability to export the components identified for use by other tools, or to import an initial component description of the system.

An important prerequisite for the value of the ISVis analyses is the choice of usage scenarios with which to exercise the subject system. The particular event traces that are examined directly affect the efficiency with which an analyst can localize behavior. Another shortcoming is the complexity of the user interface. While we have made every effort to make ISVis user-friendly, there is a marked trade-off between the powerful features available to the analyst and the ease with which these features can be learned. A usability study and future public release will shed some light in this area.

There are other program understanding approaches that accomplish some of the goals of the case study, even as simple as examining a static call graph or "grep'ing" source code for particular identifiers. For example, when the analyst began using XEmacs to view source code while constructing component mappings, he or she could have taken advantage of other tools which display calling relationships graphically. Ultimately, it will be a combination of tools and techniques that help an analyst understanding programs. The ISVis approach is another useful technique specifically aimed at understanding the behavior of a program, its actors and interactions and scenarios that purely static understanding techniques cannot hope to provide.

5. Implications

The success of ISVis on this case study raises several issues related to its future development and applicability. Among these are its scalability, its extensibility, and its interoperability with other tools.

5.1 Scalability

Architectural extraction and localization are interesting problems only if the system being analyzed is sufficiently large that an architectural overview is required to convey understanding. Consequently, it is important for extraction and localization technology to scale up to large systems.

Scaling has several implications to ISVis. On the positive side, the abstraction methods provided by ISVis allow it to deal with arbitrarily complex systems. On the negative side, however, limited machine resources can intrude on its success. For example, many visualizations often have a problem scaling to display large amounts of data. ISVis' Information Mural has proven effective at condensing large amounts of data and enabling efficient access by the analyst.

Dealing with large amounts of event trace data is also a resource problem. In principle, a small program can generate an infinite amount of trace data. Hence, it is important for the analyst to determine the degree of resolution in the event trace necessary to capture essential connectors without overwhelming storage and computational resources.

5.2 Extensibility

ISVis has been designed to be extensible in two ways. First, the underlying classes of actors with which it can deal is not dependent on a particular programming language. In fact, if appropriate static analysis tools are available, ISVis can use any primitive set of actors. For example, ISVis was originally designed to analyze program written in the C++ language. However, Mosaic is written in C. In this case, using ISVis was simply a matter of ignoring class actors, which, in any event, generate no instances when C programs are analyzed. Moreover, the generality also works in the other direction. New actors, such as modules or subsystems, can be added without effecting the underlying ISVis process.

The second form of extensibility has to do with event traces. The only classes of events used in the Mosaic case study were function calls and returns. But, as far as ISVis is concerned, events have types, and the exact nature of the type is unimportant to the pattern matching ISVis provides. For example, ISVis would be quite able to use task invocation or synchronization as event types.

5.3 Interoperability

ISVis is designed to interoperate with other tools. Two specific examples can be given. The first has to do with the problem of detecting connectors. A useful connector is one that characterizes common component interactions. But the abundance of event data from log files makes determining commonality difficult. Internally, ISVis supports several straightforward heuristics to detect patterns. However, other tools are available that may do a more sophisticated job at this task. In particular, we are looking at the Balboa tool [3]. Balboa is capable of applying several machine-learning techniques to the problem of describing complex event traces. Balboa produces as output a finite state machine that is capable of generating the event sequence. To the extent that the machine and its corresponding regular language are much smaller than the event trace, they provide a candidate abstraction for the ISVis analyst to use. We intend to hook Balboa to the ISVis pattern matcher and see how this enhancement extends the power of ISVis analyses.

The second form of interaction we intend to investigate is with an architectural analysis tool called SAAMTool. SAAM [11] is an architectural analysis method that uses usage scenarios to guide analysts in making decisions about the desirability of a proposed enhancement to a software system. SAAMTool supports this process by, among other things, providing a graphical display tool for architectures. SAAMTool itself does not generate architectural models, so it is natural to use ISVis for that purpose. We are currently looking at ACME [7] as an interoperability mechanism for the architectural models that need to be communicated between ISVis and SAAMTool.

6. Related Work

Several different areas overlap with our work, including software visualization, program understanding, and reverse engineering. 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) [2]. They have built tools to display and edit TMFDs, to generate TMFDs from event traces, and to simulate the operation of 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.

Sefika, Sane, and Campbell have done work in architectural visualization of systems with goals similar to ours [17]. Their views seek to portray the operation of a system from various architectural levels, and they have developed an unobtrusive instrumentation system to efficiently gather event trace data. However, some of their views are tightly coupled to the domain of the subject system rather than generic to software architectures—possibly because their subject program is an operating system.

The notion of a *design pattern* as a solution to a problem in a particular context provides a literary form through which software design experience can be documented to be reused by others [4][6]. Similarly, our interaction patterns are so named because they too are repeatable entities and because they create visual patterns on the screen. The two types of patterns reinforce each other because interaction patterns result from instances of design patterns and can be seen as low-level evidence for their existence.

Murphy, et al. have developed an approach that allows software engineers to specify a high-level model of a system and how the source code maps into that model [14]. 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 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 sequential, behavioral models.

The Program Explorer is a C++ program understanding tool that is focused on class and object centered views [12]. 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. The system is designed to execute the program for a while, stop execution, and then focus on particular classes or objects. It is not intended as a global understanding tool, so the users must know what (or where in the execution) they are interested in before they start. Examples of using the system to uncover design patterns in real-world sized systems are given.

The OO!CARE tool is the C++ version of the CARE environment for C program understanding [13]. 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.

7. Conclusion

ISVis is a method and tool for analyzing software for purposes of modeling its architecture. It combines static and dynamic analyses to determine the software's components and connectors. The ISVis tool supports this process with graphical views capable of displaying large amounts of interaction data and for making abstractions over them. We have applied the tool to a real-world problem, extending the Mosaic web browser, and it provided significant support for the task. We are continuing to apply ISVis in other case studies to help evaluate its usefulness. These include an examination of the view redraw mechanisms used in typical GUI applications and also a case study involving the ISVis tool examining itself. The experience reinforces our belief that architectural understanding requires both static and dynamic information to be truly valuable.

Acknowledgments

Effort sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-2-0229. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[1] Gregory Abowd, Ashok Goel, Dean F. Jerding, Michael McCracken, Melody Moore, J. William Murdock, Colin Potts, Spencer Rugaber, and Linda Wills. “MORALE/Mission Oriented Architectural Legacy Evolution.” To appear in the *Proceedings of the International Conference on Software Maintenance’97,* Bari, Italy, September 29-October 3, 1997.

[2] Wayne Citrin, Alistair Cockburn, Jurg von Kanel, and Rainer Hauser. “Using Formalized Temporal Message-Flow Diagrams.” *Software—Practice and Experience*, 25(12): 1367-1401, December 1995.

[3] Jonathan E. Cook and Alexander L. Wolf. “Automating Process Discovery through Event-Data Analysis.” *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, April 1995.

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

[5] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo. “A Cliche-Based Environment to Support Architectural Reverse Engineering.” *International Conference on Software Engineering,* Monterey, California, November 4-8, 1996, 319-328.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Software*. Addison-Wesley, 1995.

[7] D. Garlan, B. Monroe, and D. Wile. “ACME: An interchange language for software architecture, 2nd edition.” Technical report, Carnegie Mellon University, 1997.

[8] David R. Harris, Alex S. Yeh, and Howard B. Reubenstein. “Extracting Architectural Features from Source Code.” *Automated Software Engineering*, 3(1/2):109-138, June 1996.

[9] Dean F. Jerding 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,* Atlanta, Georgia, October 1995, pp. 43-50.

[10] Dean F. Jerding, John T. Stasko, and Thomas Ball. “Visualizing Interactions in Program Executions.” To appear in the *Proceedings of the International Conference on Software Engineering*, 1997.

[11] R. Kazman, L. Bass, G. Abowd, and S. M. Webb. “SAAM: A Method for Analyzing the Properties of Software Architectures.” *Proceedings of the International Conference on Software Engineering 16*, Sorrento, Italy, May 1994, 81-90.

[12] Danny B. Lange and Yuichi Nakamura. “Interactive Visualization of Design Patterns Can Help in Framework Understanding.” *Proceedings of ACM OOPSLA '95,* 1995, pp. 342-357.

[13] P. K. Linos and V. Courois. “A Tool for Understanding Object-Oriented Program Dependencies.” *Proceedings of the Workshop on Program Comprehension*, 1994, pp. 20-27.

[14] G. C. Murphy, D. Notkin, and K. Sullivan. “Software Reflexion Models: Bridging the Gap Between Source and High-Level Models.” *Proceedings of the Foundations of Software Engineering*, 1995.

[15] National Center for Supercomputing Applications. “NCSA Mosaic Home Page.” http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html.

[16] Marshall T. Rose. *The Internet Message: closing the book with electronic mail.* Prentice-Hall, ISBN 0-13-092941-7.

[17] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. “Architecture-Oriented Visualization.” *Proceedings of ACM OOPSLA ‘96*, 1996, pp. 389-405.