Extraction of Architectural Connections from Event Traces

Dean Jerding, Spencer Rugaber[1]
Georgia Institute of Technology
{dfj, spencer}@cc.gatech.edu

Abstract

An important part of understanding a program is obtaining a representation of its architecture. One way of viewing architectures is in terms of their components and connectors. While static program analysis is useful for extracting components from a program's source code, connectors are more problematic. This paper describes a tool we have developed and an another we have adapted in order to support an analyst in visualizing, abstracting, and inferring architectural connectors in programs.

Keywords: Software architecture, event trace, reverse engineering, program understanding, grammatical inference, software visualization

1. Motivation

Most software development takes the form of enhancements to existing programs. Consequently, understanding existing programs is a central activity to most software developers. Program understanding takes many forms from the reading of individual program statements in order to fix a bug to viewing the overall system structure in terms of its major pieces. The latter activity is concerned with understanding a system's software architecture.

Two aspects of software architecture are central to program understanding—*components* and *connectors*. "Components are the loci of computation and state" [14]. As such, they correspond to discernible units in the program text, such as statements or functions or object classes. This enables tools to extract a description of a program's components by using static program analysis (examination of a program's source code).

"Connectors are the loci of relations among components. They mediate interaction but are not things to be hooked up (rather, they do the hooking up)" [14]. This suggests that static analysis may be limited in its effectiveness in extracting connectors. Instead, we propose to use dynamic analysis (examination of actual program executions) to extract architectural connectors.

An *event trace* is a form of dynamic data that comprises a log of program activities. Such activities are generically called *events*, and they may be visualized using *event trace diagrams* [13] (also called message sequence diagrams, temporal message flow diagrams [5], or interaction diagrams). An example of an event trace diagram is shown in Figure 1.

An event trace diagram consists of a series of vertical columns, each corresponding to a system component, and horizontal lines connecting two or more columns. Each line denotes an event that took place among the components in the associated columns. In Figure 1, each event corresponds to a function call or return from the component at the end of the line containing the white dot to the component at the other end of the line. Events are time-ordered with events at the bottom of the diagram occurring after events at the top.

Because connectors "mediate interaction" and event trace diagrams depict a class of interaction, it is appealing to consider trying to extract meaningful interactions from the diagrams. In particular, we will identify connectors with *interaction patterns* [9], denotations of recurring interactions, and try to detect interaction patterns in the event trace that correspond to connectors meaningful at the software architecture level.

2. Approach

We have used three techniques in our exploration of this problem: visualization, abstraction, and inference. Event logs contain high volumes of data; even small programs can generate hundreds of thousands of events. Consequently exploring this data is essential to detecting interesting interaction patterns. We have developed a novel software visualization technique—the Information Mural [9]—to support this activity. The Mural enables an analyst to view the entire program execution history at once, searching for recurring patterns that may correspond to meaningful interactions among components. An Information Mural can be seen along the right edge of the event

1. Point of contact: spencer@cc.gatech.edu

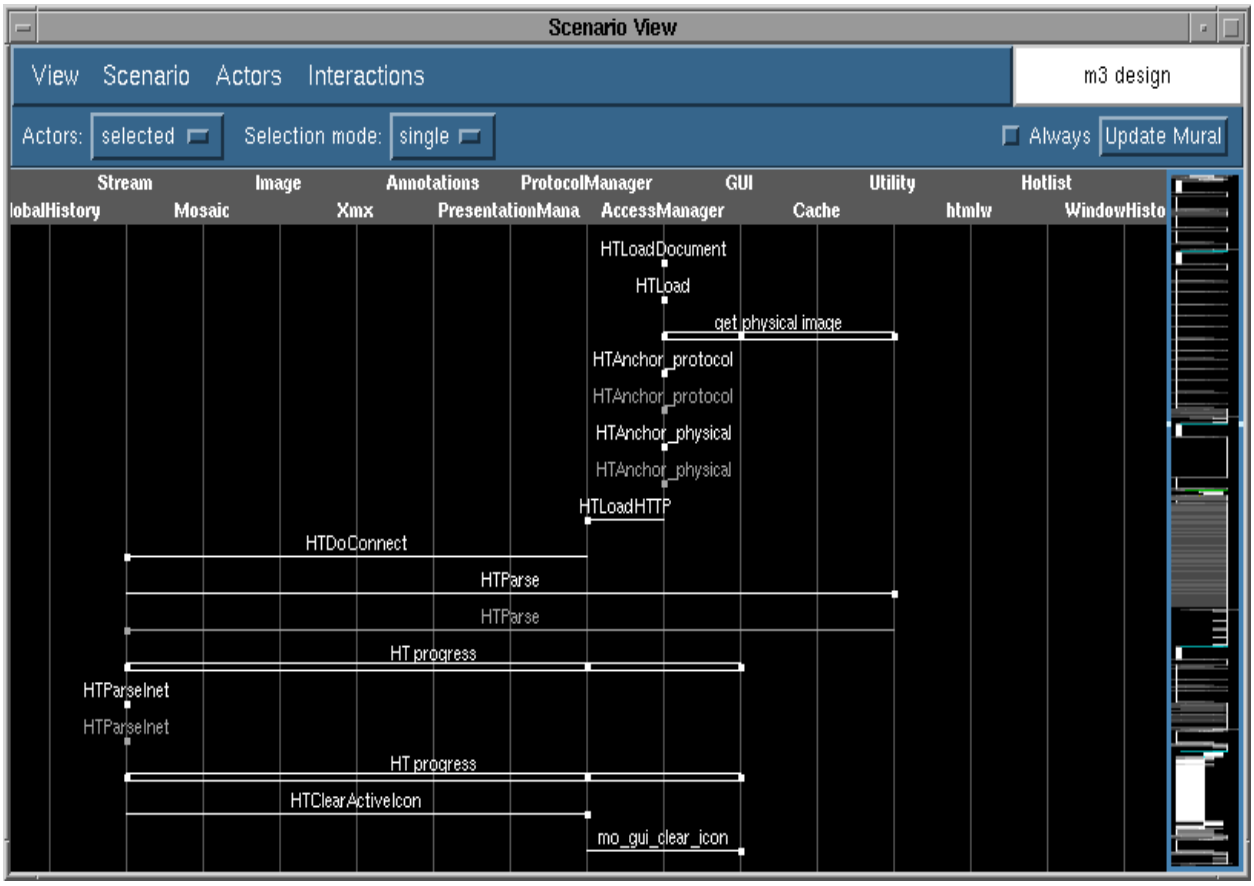

Figure 1: Event Trace Diagram

trace diagram in Figure 1. Information Murals are described in more detail in Section 3.

Once the event log can be effectively visualized, the analyst is able to construct higher-level models of it. The ISVis (Interaction Scenario Visualizer) tool has be developed to support this process [9]. ISVis aids an analyst in constructing abstractions of both components and connectors. ISVis is described in more detail in Section 4.

ISVis is capable of detecting repeated occurrences of simple interaction sequences. However, interesting system behavior is manifested by more complex inter-component interactions. Consequently, we would like to be able to infer more complex patterns. To this end, we have connected ISVis to the Balboa tool-set developed at the University of Colorado [6]. Balboa supports *grammatical inference*, the construction of a grammar that explains a stream of events taking place among system components [4][11]. The grammars produced by Balboa serve as a description for the behavior of connectors between those components. Grammatical inference and Balboa are described in more detail in Section 5.

3. The Information Mural

The Information Mural technique allows two-dimensional visual representations of large information spaces to be created even when the number of informational elements greatly outnumbers the available pixels. Current methods for depicting such large information spaces typically utilize abstraction, over-plotting, or sampling to create a view of the entire space. Alternatively, scrollbars are used to allow access to different parts of the information. All of these techniques result in a loss of information that might be useful to the observer.

The Information Mural technique increases the visual information bandwidth available to visualization applications. An Information Mural is a two-dimensional, miniature representation of an entire information space that uses visual attributes such as color and intensity along with an anti-aliasing-like compression technique to portray attributes and density of information. The goals of the visualization technique can be summarized as follows:

- to create a representation of an entire (large) information space that fits completely within a display window or screen;

- to mimic what the original visual representation of the information looks like when it is viewed in its entirety;
- to minimize the loss of information in the view, regardless of the size of the compressed representation.

3.1 Technique

Imagine some visual representation of a large information space, made up of distinct elements each with their own representation. If an Information Mural of this space is to fit in some area of **`I x J`** pixels; assume there is a "bin" associated with each pixel. The position of each information element is first scaled to fit into the available space. As each element is "drawn" in the mural using an imaginary pen, different amounts of "ink" fall into different bins. As each subsequent element is drawn, the amount of ink builds up in different bins, depending on the amount of overlap of the elements.

The resulting Information Mural is created by mapping the amount of ink in each bin (the information density) to some visual attribute. In a grayscale mural, the shade of each pixel corresponds proportionally to the amount of ink in each bin. Instead of using grayscale variation, an equalized-intensity variation over the entire color scale can be used. With a raindrop mural, for example, the amount of ink in each bin makes a "puddle" centered around that pixel, so pixels with more ink appear larger. Color can then be added to the mural to convey other attributes of the informational elements, while still preserving the density mapping.

3.2 Implementation

The Information Mural is implemented as an abstract widget which can be used by an application just like a scrollbar, drawing area, or other graphical widget. The widget can be used purely for output, to display an Information Mural. Or it can act as a global companion view to a more detailed view by providing a "navigation rectangle" which can be panned and zoomed by the user. The widget is written in C++ on top of X Windows and Motif. The `Mural` class provides a basic application interface to create, lay out, and draw a mural. Client applications inherit from the `Mural_Client` class to receive interaction notification methods that the application may choose to implement.

When an instance of a Mural is created, the application defines the coordinate system in which the Informa-

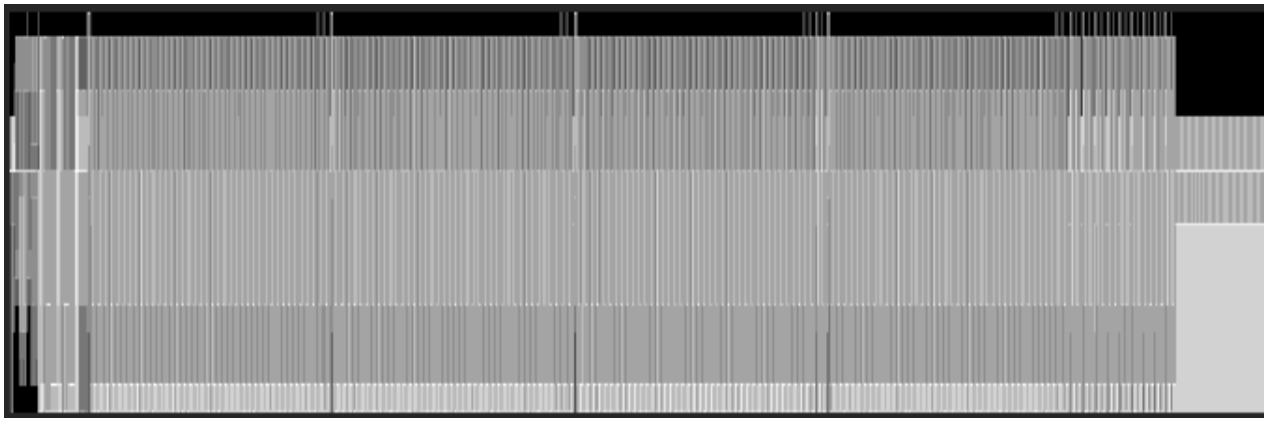

Figure 2a: Mural of an event trace of over 50,000 messages, drawn in an area 500 pixels wide

Figure 2b: Same diagram drawn by just over-plotting (without using the Information Mural technique)

tion Mural is drawn. If the Mural's navigation capabilities are to be used, the initial position and size of the navigation rectangle must also be set. All of the drawing methods (`MuralDrawPoint()`, `MuralDrawLine()`, `MuralFrame-Rectangle()`, etc.) are passed coordinates in the application-defined coordinate system. Whenever the Mural needs to be redrawn, it calls the application's `MuralRedrawNeededCB()` callback method. Additionally, whenever the navigation rectangle is moved or the Mural is zoomed into by the user, the application's `MuralValueChangedCB()` and `MuralZoomedCB()` are called, respectively.

In this way, the application draws the Information Mural in its own coordinate space with respect to the information being displayed, and the Mural widget handles the rendering of the mural in whatever space it has on the screen. User interactions with the Mural widget are passed back to the application in the application-defined coordinate space as well.

3.3 Conclusion

Figures 2a and 2b displays the same data with and without using the Information Mural algorithm. Although Figure 2b may look as though it contains mostly white space, this is actually an indication that nearly all of the pixels in the region are white. Figure 2a gives a much more informative view of the original information.

We have examined event traces containing as many as one million events. Trying to detect interesting patterns of information in such a display would have been impossible without a visualization technique such as the Information Mural. Once patterns are detected, then the ISVis tool can be used to abstract away details, thereby providing opportunities to detect other interesting interaction patterns.

4. ISVis

ISVis (Interaction Scenario Visualizer) is a tool that uses abstraction to support the program understanding process. Program executions are made up of recurring patterns of interactions, manifested as repeated sequences of program events such as function calls, object creations, and task initiations. 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 support of all three of these activities.

The purpose of ISVis is to support the process of abstracting high-level static and dynamic structures from program text and event traces. 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.

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

ISVis assumes the existence of an independent static analysis tool. In its current implementation, it makes use of information provided by the Source Browser facility that accompanies SUN's Solaris C and C++ compilers. The static analyzer reads the Source Browser database files and generates a static information file. An instrumentation tool that accompanies ISVis takes the source code, the static information file, and information supplied by the analyst about what parts of the code to instrument, and generates instrumented source code. This code must be compiled externally to the ISVis tool, and then, when the instrumented system is executed using relevant test data, event traces are generated and read into ISVis. The user then interacts with ISVis to abstract the source code into *actors* and the event traces into *scenarios*.

ISVis currently provides two views, the Main View and the Scenario View. Figure 3 is a snapshot of the Main View. The top portion of the view lists the actors, including files, classes, functions, and user-defined aggregations of them. The middle portion of the Main View includes lists of the scenarios and interactions, 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 small window for textual information entry and display. Note that each of the scrollable lists of actors and interactions uses an Information Mural 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 the second ISVis view, a Scenario View, for each scenario in the model. Figure 1 shows a Scenario View. The Scenario View is in fact an 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. This allows the analyst to observe various phases in the

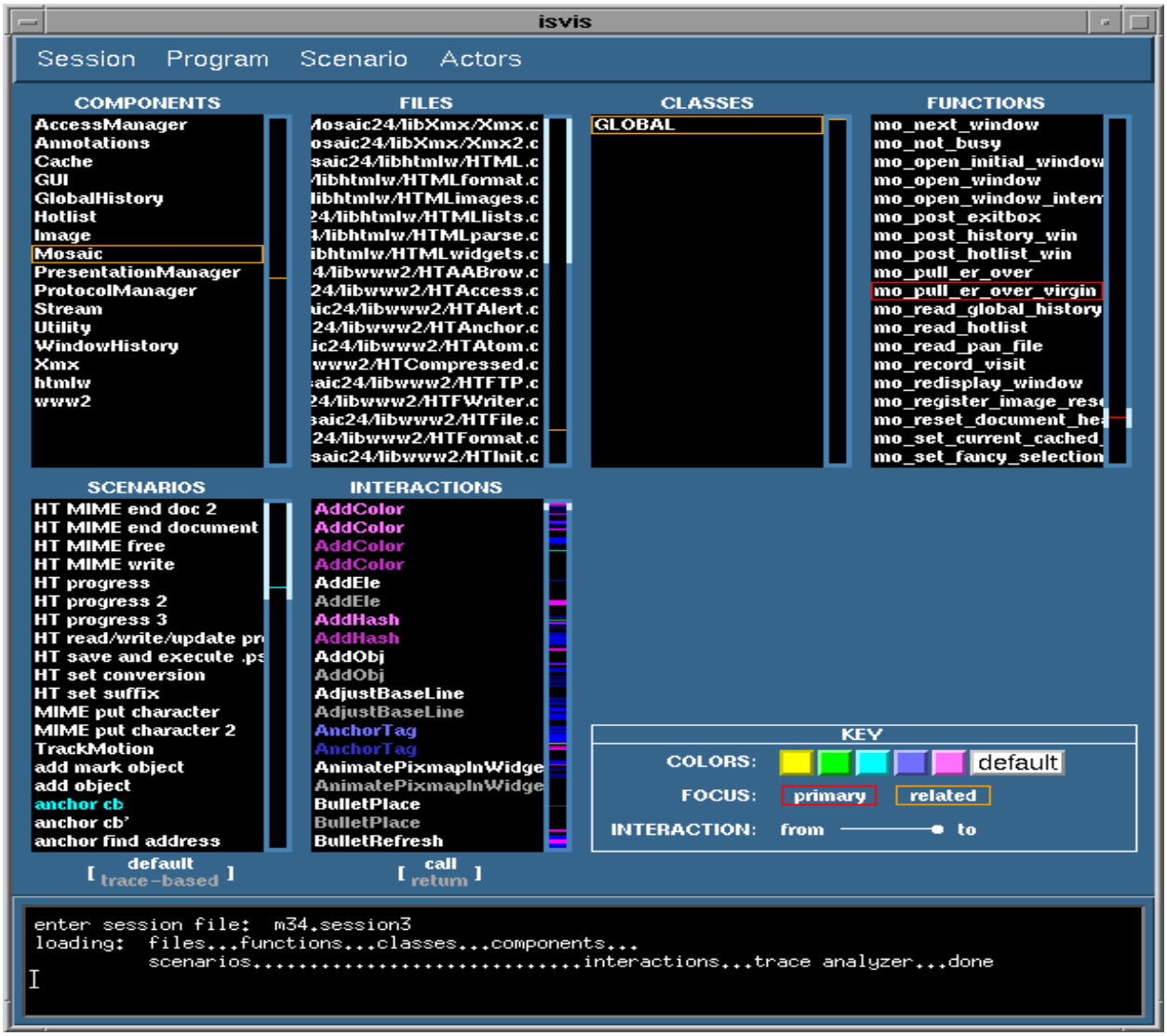

Figure 3: ISVis Main View

scenario including repetitive visual patterns indicating the presence of frequently occurring interactions in the 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, and then all occurrences of that sequence of interactions in the original event trace are replaced with a reference to the 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 slim, horizontal rectangle containing all of the actors involved in the scenario.

The Scenario View also provides a feature 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 final pattern feature includes the ability to ask ISVis to look for repeated

sequences of interactions that occur in the event trace. 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 such that any selection or modification done in one view is immediately reflected in the other. Also, it is possible to save the current session status for later analysis.

5. Grammatical Inference and Balboa

5.1 Grammatical Inference

"Grammatical Inference encompasses theory and methods for learning grammars from training data." [16] Although many different types of grammars have been used, the most successful and commonly used grammar is the type three or regular grammar. These grammars are normally expressed using regular expressions of finite state machines.

Any stream of events can be trivially modeled with a state machine. Imagine, for example, a stream of fifty events. Construct a state machine containing exactly fifty states, one for each event occurrence in the input stream. Each state is linked only to its successor by a transition on the corresponding event in the input stream. Not only is this solution inelegant, but it explains only the input stream and is of no use in predicting either the next element in the stream or elements in another stream from the same source. The solution is too specific.

Another trivial solution is a non-deterministic finite state machine containing exactly one state. There are as many transitions as there are different input events, and all transitions lead back into the single state. Although this solution also models the input stream, it is no more satisfactory than the first. Its major problem is that it explains not only the given input stream, but every other input stream as well. It is too general.

The goal of grammatical inference, therefore, is to find a grammar that is of the right size and is capable of predicting or modeling not only the input stream, but other streams from the same source. There are two implications of this: that any solution will be an approximation, trading off conciseness for predictive power, and that a suitable solution mechanism should include parameters to facilitate tuning the grammar until a desired balance can be found.

5.2 Inference Mechanisms

Many different approaches have been taken to the problem of grammatical inference, such as neural networks and genetic algorithms. The two that Balboa uses are hidden Markov models [2] and `k-tail` algorithms [10]. A Markov process is similar to a state machine in which transitions are labeled with the probability of the corresponding transition occurring. Markov models are useful for modeling event streams produced by stochastic processes. A hidden Markov model is one in which the parameters of the Markov model must be inferred from its behavior.

A `k-tail` grammatical inferencing algorithm is one in which an overly specific state machine, such as the first unsatisfactory solution described above, is successively refined by merging states. Two states are merged if their next `k` states are identical. Of course, `k` serves as a tuning parameter for this approach.

5.3 Balboa

Balboa is a tool developed by Jonathan Cook and Alex Wolf at the University of Colorado [6]. They have used it to model organizational processes and validate that an actual process corresponds to an intended one. Their research included the exploration of various approaches to grammatical inferencing including Markov modeling, neural networks, and `k-tail` algorithms.

The interface to the Balboa tool suite is provided by Tcl/Tk [12] and consists of tools for managing event streams, identifying the salient features of events, invoking the two grammatical inference methods, and displaying the resulting state machine using the `dotty` tool from AT&T [7].

To use Balboa, we took an ISVis event stream, converted it to a format suitable for input to Balboa, and asked Balboa to produce a state machine model. An example of the models produced is shown in Figure 4. In this case, relevant events included calls and returns between functions in the systems being analyzed. We looked at both an entire execution trace and projections of it that pertained to a specific component of the program.

6. Status

ISVis has just completed its beta test evaluation at three sites in the U.S. and Europe. It will be made available for public release as soon as the results of the evaluation can be addressed. The ISVis web page can be found at URL `http://www.cc.gatech.edu/morale/tools/isvis/isvis.html`. Balboa can be obtained on the Internet from URL `http://www.cs.colorado.edu/~serl/process/Balboa`.

Our next step is to connect ISVis directly to Balboa. This will provide several benefits. First, abstractions recognized by an analyst using ISVis can be fed to Balboa, thereby guiding its inference process. Furthermore, interesting patterns detected by Balboa can be proposed to the ISVis analyst for examination using the Information Mural and the ISVis view windows. We envisage this process being iterative, with the analyst switching back and forth between the tools.

7. Discussion

The use of dynamic analysis to detect software architectural connections raises several issues including the formal power of the grammars used to describe connector abstractions and the fidelity of the resulting models.

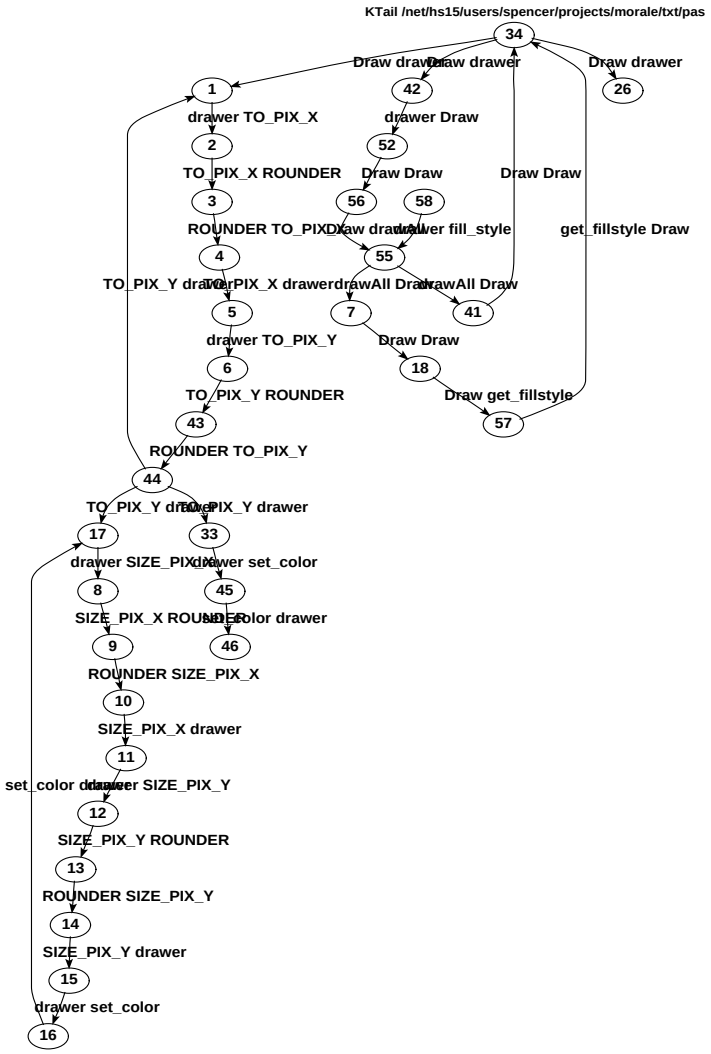

Figure 4: State machine produced by Balboa using k-tail algorithm

7.1 Representing Connector Abstractions

The physical part of the ISVis-Balboa connection is straightforward—either temporary files or UNIX pipes can be used. More interesting is how to encode abstract interactions. Unlike abstract component descriptions that are made up of lower-level components, lower-level connections combine using a protocol specifying conditions under which the interactions can interleave.

ISVis patterns express the sequential occurrence of events in an input stream. Because the ISVis pattern matcher can detect repeated occurrence of such sequences, it is capable of expressing patterns roughly equivalent to path expressions [3]. The qualification is due to the fact that one form of ISVis pattern matching (the interleaved match) allows arbitrary, non-pattern events to be interleaved with the events being searched for. Balboa, on the other hand, is capable of inferring regular expressions denoted by finite state machines. Hence, Balboa can provide suggestions to the analyst that ISVis cannot.

However, what we are really interested in detecting are architectural connections. Although there are many different ways to express these, the Wright language is typical [1]. In Wright, descriptions of connections include not only structural information but also a formal semantic notation for describing the interaction behavior of the connection. The semantics are expressed in a process algebra notation, derivative of CSP [8]. Because CSP can describe the interleavings of independently executing processes, event streams from these processes cannot be described by regular expressions. In fact, the streams are examples of *shuffle grammars*, which are not even context free. The implication is that fully automated detection of architectural connections is theoretically difficult. For a related use of CSP for describing occurrences of atomic events, refer to [15].

7.2 Approximations

The typical mechanism for designers to communicate the architecture of a system is to use a single, high-level "box-and-arrow" diagram. Their popularity indicates that they are useful for communicating abstractions, but they suffer from several difficulties. One difficulty is due to their informality. This usually takes the form of multiple or vague interpretations being given to the arrows in the diagrams. Research into software architectures and architectural description languages are addressing this problem. However, even a formally specified diagram is still only an approximation. That is, it is someone's idea of what is important and what is not important about a system. But different system aspects are important to different people. And therefore multiple high-level architecture diagrams may be required. For example, some people might be interested in how a system is structured to guarantee its performance requirements while others may be interested in how it will provide reliability.

A further problem is due to *design drift.* This occurs when the original, presumably clean structure of a system disintegrates over time due to subsequent maintenance activities. It is here that having an automatically-derived description of a system's actual components and connectors to contrast with the ideal one shown in the box-and-arrow diagram can be of value.

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.This work was supported

References

[1] Robert Allen and David Garlan. "A Formal Basis for Architectural Connection." *ACM Transactions on Software Engineering and Methodology*, 6(3):213-249, July 1997.

[2] Reinhard Blasig. "Discrete Sequence Prediction with Commented Markov Models." in *Grammatical Inference: Learning Syntax from Sentences*, Laurent Miclet and Colin de la Higuera, editors, Springer-Verlag, *Lecture Notes in Artificial Intelligence*, No. 147, 1996, pp. 191-202.

[3] R. H. Campbell and A. N. Habermann. "The Specification of Process Synchronization by Path Expressions." *Proceedings of Operating Systems, Lecture Notes in Computer Science*, Volume 16, pp. 89-102, Springer Verlag, 1974.

[4] Rafael C. Carrasco and Jose Oncina, editors. *Grammatical Inference and Applications*, Springer-Verlag, Lecture Notes in Artificial Intelligence, 862, 1994.

[5] 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.

[6] 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.

[7] E. R. Gansner, E. Koutsofios, S. C. North, K. P. Vo, "A Technique for Drawing Directed Graphs," *IEEE Transactions on Software Engineering*, 19(3):214-230, 1993.

[8] C. A. R. Hoare. "Communicating Sequential Processes." *Communications of the ACM*, 21(8): 666-677, August 1978.

[9] Dean Jerding and Spencer Rugaber Using Visualization for Architectural Localization and Extraction." *Proceedings of the Fourth Working Conference on Reverse Engineering*, Amsterdam, the Netherlands, IEEE Computer Society, October 6-8, 1997.

[10] Timo Knuutila. "Inductive Inference from Positive Data: From Heuristics to Characterizing Methods." in *Grammatical Inference: Learning Syntax from Sentences*, Laurent Miclet and Colin de la Higuera, editors, Springer-Verlag, Lecture Notes in Artificial Intelligence, No. 1147, 1996, pp. 22-47.

[11] Laurent Miclet and Colin de la Higuera, editors. *Grammatical Inference: Learning Syntax from Sentences.* Springer-Verlag, Lecture Notes in Artificial Intelligence, No. 1147, 1996.

[12] John K. Ousterhout. *Tcl and Tk Toolkit*. Addison Wesley, 1994.

[13] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[14] Mary Shaw and David Garlan. *Software Architecture / Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[15] Andrew M. Tyrrell and Geof F. Carpenter. "CSP Methods for Identifying Atomic Actions." *IEEE Transactions on Software Engineering*, 21(7):629-639. July 1995.

[16] Enrique Vidal. "Grammatical Inference: An Introductory Survey." *Grammatical Inference and Applications*, Rafael C. Carrasco and Jose Oncina, editors, Springer-Verlag, 1994, *Lecture Notes in Artificial Intelligence*, No. 862, 1-4.