Appeared in *Proceedings of the Conference on Object-Oriented Programing, Systems, Languages, and Applications* (Vancouver, British Columbia, Canada; 18–22 October 1998), ACM SIGPLAN, pp. 271–283, 1998. Published as ACM SIGPLAN Notices, **33**(10), October 1998.

Visualizing Dynamic Software System Information through High-level Models

Robert J. Walker and Gail C. Murphy
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC, Canada V6T 1Z4
{walker, murphy}@cs.ubc.ca

Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak
Object Technology International, Inc.
301-506 Fort St.
Victoria, BC, Canada V8T 1X3
{bnfb, Darin_Wright, Darin_Swanson}@oti.com
jeremyi@unixg.ubc.ca

ABSTRACT

Dynamic information collected as a software system executes can help software engineers perform some tasks on a system more effectively. To interpret the sizable amount of data generated from a system's execution, engineers require tool support. We have developed an off-line, flexible approach for visualizing the operation of an object-oriented system at the architectural level. This approach complements and extends existing profiling and visualization approaches available to engineers attempting to utilize dynamic information. In this paper, we describe the technique and discuss preliminary qualitative studies into its usefulness and usability. These studies were undertaken in the context of performance tuning tasks.

Keywords

Software visualization, programming environments, software structure, program comprehension, execution trace, performance.

1 INTRODUCTION

Effective performance of many software engineering tasks requires knowledge of how the system works. Gaining the desired knowledge by studying or statically analyzing the source code can be difficult. Static analysis, for instance, can help a software engineer determine if two classes can interact, but it does not help the engineer determine how many objects of a class might exist at run-time, nor how many method calls might occur between particular objects. Determining answers to these questions requires an investigation of dynamic information collected as the software system executes. This dynamic information helps bridge "the dichotomy between the code structure as hierarchies of classes and the execution structure as networks of objects" [1, p. 326].

Software engineers require tool support to effectively access and interpret dynamic system information, because the quantity, level of detail and complex structure of this information would otherwise be overwhelming. In creating a tool to help an engineer access this information, two goals must be paramount: the tool must be usable and it must be useful for the task it is designed to address. Usability is defined in terms of practicality and simplicity of interface; usefulness is defined in terms of easing the performance or completion of a task of importance, especially in comparison to alternative methods.

Many tools have been developed to provide engineers access to dynamic information. Profilers, for instance, provide numerical summaries of dynamic information, such as the length of time spent executing a method. This information can be helpful when trying to tackle some system performance problems. Other tasks, however, such as verifying that objects are interacting appropriately according to defined roles [6], require additional structural information. The usefulness of profilers for these types of task degrades because the relevant dynamic information is not evident from a summary numeric value produced on a per method or per class basis.

When structural dynamic information is needed, an engineer may attempt to use an object-level visualizer (e.g., [1, 6, 7]). These visualizers provide such displays as the interactions between objects (or classes) and the number of objects created of each class. When the task requires views involving many classes in a large system, the usability of these tools degrades, as they tend to display complex interactions between multiple objects in a haze of extraneous, overlain information.

In part to overcome this complexity problem, Sefika *et al.* introduced an architectural-oriented visualization approach [14] that allows an engineer to investigate the operation of the system at both coarse- and fine-grained levels. Some of the design choices made in their approach limit its applicability. Their approach is on-line, limiting its usefulness for some kinds of tasks. Their approach requires hard-wired instrument classes to be attached to the system, limiting its flexibility and reducing its usability.

We have developed an off-line, flexible approach for visualizing the operation of an object-oriented system at an architectural level. Our approach abstracts two fundamental pieces of dynamic information: the number of objects involved in the execution, and the interactions between the objects. We visualize these two pieces of information in terms of a high-level view of the system that is selected by the engineer as useful for the task being performed.

To represent the information collected across a system's execution, we use a sequence of *cels*. Each cel displays abstracted dynamic information representing both a particular point in the system's execution, and the history of the execution to that point. The integration of "current" and "historical" information is intended to ease the interpretation of the display by the engineer. Using our prototype, a software engineer can navigate both forwards and backwards through the cels comprising views on the execution.

Our approach complements and extends existing approaches to accessing dynamic system information. Our approach

- allows an unfamiliar system to be studied without alteration of source code,
- permits lightweight changes to the abstraction used for condensing the dynamic information,
- supplies a visualization independent of the speed of execution of the system being studied, and
- allows a user to investigate the abstracted information in a detailed manner by supporting both forwards and backwards navigation across the visualizations.

To investigate the usefulness and usability of the approach, we have performed preliminary, qualitative studies of the use of the technique to aid performance-tuning tasks on Smalltalk programs. These studies show that the technique can help software engineers make better use of dynamic system information when performing tasks such as performance enhancement.

We begin in Section 2 by describing our visualization technique; Section 3 discusses the creation of a visualization. In Section 4, we discuss our initial evaluation efforts intended to assess the usability and usefulness of the approach. In Section 5, we consider the design choices we made in our visualization technique. Section 6 describes related work and Section 7 concludes with a description of directions for future work.

2 VISUALIZATION TECHNIQUE

Our visualization technique abstracts information that has been previously collected during a system's execution and uses concepts from the field of computer animation to display that information to a user. We begin the description of our technique by focusing on the visualization itself, and then describe how a software engineer can construct such a visualization.

Figures 1 through 4 show different views within a visualization produced during one of our case studies that was investigating a performance problem in a reverse engineering program (Section 4.1). The two windows in Figures 1 and 2 each provide one view—a cel showing events that occurred within a particular interval of the system's execution, defined as a set of n events where n is adjustable. The view in Figure 3 shows a summary view of all events occurring in the trace, and Figure 4 gives a detailed, textual view of some of the information within the summary view. Sections 2.1, 2.2, and 2.3 describe these views in more detail. Full details of the running example we use are provided in Section 4.1.

Our prototype permits a software engineer to easily switch from a particular cel to the summary view. A user may also move through the sequence of cels sequentially or via random access; animation controls, such as play, stop, step forward and step backward, allow a user to review the execution trace and pause or return to points of interest. We discuss the navigation capabilities of our visualization in Section 2.3.

2.1 Cels

A cel consists of a canvas upon which are drawn a set of widgets. These widgets consist of:

- boxes, each representing a set of objects abstracted within the high-level model (Section 3.2) defined by the engineer,
- a directed hyperarc between and through some of the boxes,
- a set of directed arcs between pairs of boxes, each of which indicates that a method on an object in the destination box has been invoked from a method on an object in the source box,
- a bar-chart style of histogram associated with each box, indicating the ages of and garbage collection information about the objects associated with that box,
- annotations and bars within each box, and
- annotations associated with each directed arc.

Each box drawn identically within each cel represents a particular abstract entity specified by the engineer, and thus, does not change through the animation. The grey rectangles in Figures 1 through 3 labelled *Clustering*, *SimFunc*, *ModulesAndSuch*, and *Rest* are boxes corresponding to abstract entities of the same names. Two of these entities, *Clustering* and *SimFunc*, each correspond to a class in the reverse engineering tool source; the other two entities represent collections of classes.

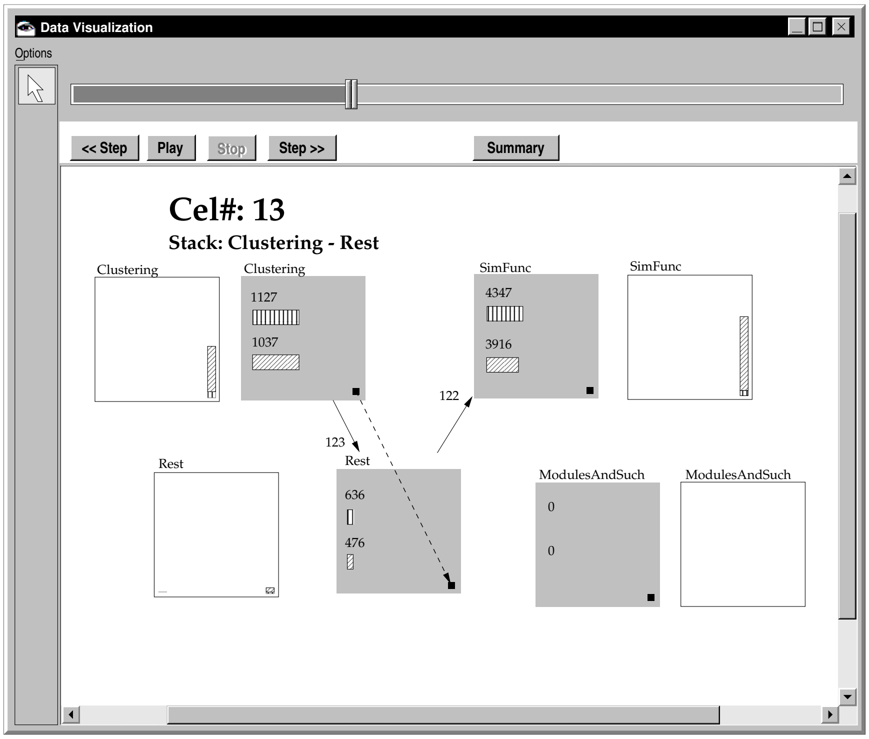

Figure 1: A window showing an example cel in the visualization technique.

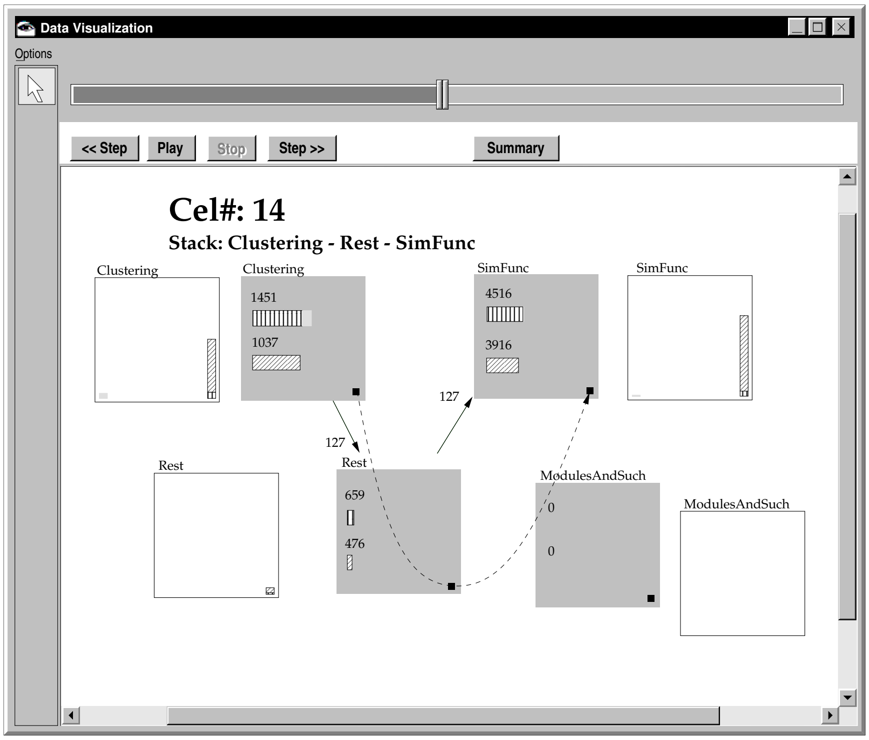

Figure 2: A window showing the next cel after that in Figure 1 for the same system and execution trace.

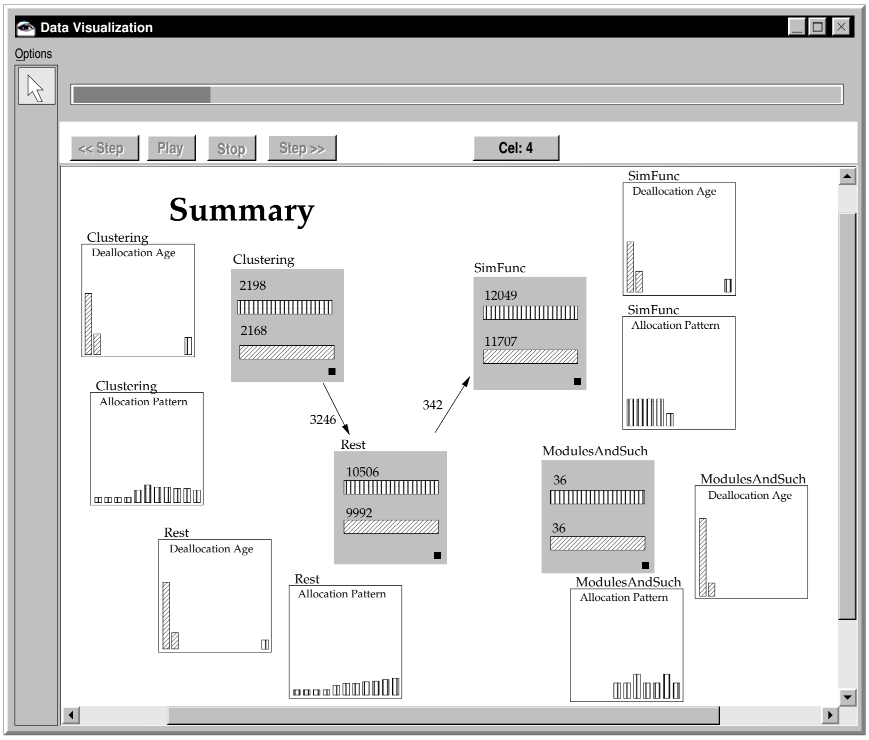

Figure 3: A window showing the Summary View for the same system and execution trace as shown in Figures 1 and 2.

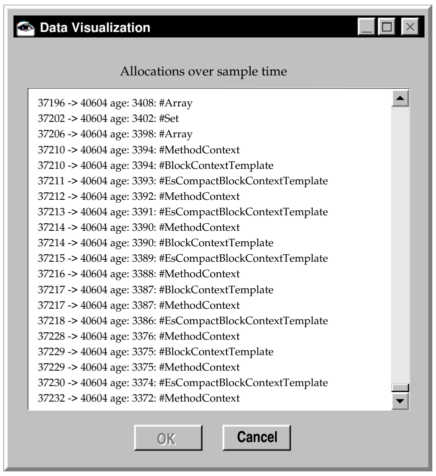

Figure 4: A pop-up window produced by clicking on the Allocation Pattern histogram for the *Clustering* entity of Figure 3.

The path of the hyperarc represents the call stack at the end of the current interval being displayed. In Figure 1, the current call stack only travels between the *Clustering* and *Rest* boxes—the hyperarc is marked in red (shown as a dashed black line herein); in Figure 2, the call stack has extended to *SimFunc* as well.

The set of directed arcs represents the total set of calls between boxes up to the current interval; they are displayed in blue (shown as solid black herein). Because the total number of pairs of boxes is manageable, this set does not obscure the rest of the cel significantly. Multiple instances of interaction between two boxes are shown as a number annotating the directed arc. The same two arcs are shown in Figures 1 and 2 from *Clustering* to *Rest*, with 123 calls, and from *Rest* to *SimFunc*, with 122 calls.

Object creation, age, and destruction are a particular focus within the visualization. Each box is annotated with numbers and bars indicating the total number of objects that map to the box that have been allocated and deallocated until the current interval. The length of a bar for a given box is proportional to the maximum number of objects represented by that box over the course of the visualization. The *Clustering* box of Figure 1 shows that a total of 1127 objects associated with it had been created to this point in the execution, and that 1037 of these had been garbage collected.

The histogram associated with each box shows this information as well, but in a more refined form. An object that was created in the interval being displayed has survived for a single interval; stepping ahead one cel, if it still exists, the object has survived for two intervals, and so on. The kth bin of the histogram shows the total number of objects mapped to the box that are of age k; to limit the number of bins in the histogram, any objects older than some threshold age T are shown in the rightmost bin of the histogram. The histogram attached to the *Clustering* box in Figure 1 indicates that all of its 1127 objects were created relatively far in the past, more than 10 intervals before the one being shown here.

Colour is used to differentiate those objects that still exist from those that have been garbage collected; each bar of the histogram is divided into a lower, green part (marked in a vertical-line pattern herein) for the living objects and an upper, red part (marked in a diagonal-line pattern herein) for the deleted objects. In Figure 1, the upper part of the bar in *Clustering*'s histogram shows that roughly 80% of the old objects have been deallocated. Yellow (shown as light grey herein) is used both within the box annotations and within histograms to indicate a change that just occurred during the interval. More specifically, it is used to show objects that have just been created or deleted. In Figure 2, which shows the interval immediately after that of Figure 1, an additional 324 objects had just been allocated that were related to *Clustering*. This allocation is shown both by the yellow (light grey) portion of the upper bar, and the yellow (light grey) bar in the first bin of the histogram.

No complex graph layout algorithms are currently used to produce the views. The drawing package used in the prototype supports interactive rearrangement of the widgets by its user.

Each cel is intended to represent a combination of information not present in its predecessor (in terms of the original execution) and a summary-to-date of the information in its predecessors and itself. The new information is difficult to interpret in isolation; the context provided by the summary-to-date eases this interpretation. See Section 5.3 for further discussion.

2.2 Summary View

In addition to the individual cels, a summary view is provided to display the overall execution of the system being studied. This view shows the same boxes, directed arcs, arc annotations, and box annotations as the final cel of the animated view. In addition, it displays two histograms per box; these are different from the histograms in the animated view. One, the *allocation pattern*, shows the entire execution trace divided into a set of ten equal-length intervals; if the trace consists of $10n$ events then each interval consists of n contiguous events. The height of each bar represents the number of objects allocated in that interval that map to that box. The other histogram, the *deallocation age*, shows the age of every object associated with the box when the object was garbage collected; if an object had not been garbage collected when tracing ended, it is displayed in the rightmost bar. For example in Figure 3, *Clustering*'s deallocation-age histogram shows that most of its objects were deallocated at a very young age while the rest still existed when tracing stopped—this is the case for all of the boxes in this example except *ModulesAndSuch*, whose associated objects were always deallocated at a young age. *Clustering*'s allocation pattern is fairly uniform, showing only a slight increase in allocations halfway through execution; on the other hand, *SimFunc* stopped allocating objects after the halfway point.

2.3 Navigating the Visualization

There are three forms of interaction with the visualization: *view selection*, *animation control*, and *detail querying*.

View selection simply entails choosing between a summary view, or the detailed, cel-based view. It would be reasonable to allow multiple, simultaneous views, both summary and cel-based, but this is not provided by the current implementation. However, the off-line nature of this technique (Section 3.1) allows multiple instances of the tool to be run simultaneously.

Animation control is provided by several buttons, a slider, and the textual entry of particular values. The buttons are Step Backwards and Forwards (by step-size number of cels), Play, and Stop. The slider is used for random access to a cel in a drag-and-drop fashion.

Textual entry is used to specify step- and interval sizes, and animation speed; altering the step size allows the engineer to move through the animation more quickly by not showing some cels—this allows the animation to proceed more quickly when the redisplay rate of the graphics software and hardware is slower than the desired rate of animation.

By default, an interval ends upon an event that caused a frame to be added or removed from the execution stack; these events include making or returning from a method call, and generally, each object allocation or deallocation. This granularity is generally too fine to be usable—with tens of thousands of method calls occurring and similar numbers of objects being created and destroyed, not much changes between two adjacent cels, and histograms tend to have empty bins except for the rightmost. Therefore, we allow the engineer to reset the interval size; a size of ten, for example, indicates that each cel should represent the changes to the system produced by ten events.

Detail querying allows the engineer to connect observations made via the abstract visualization to the actual classes, object allocations and deallocations, and method calls that are being abstracted. This is done by clicking on the appropriate widget for which details are sought. Arcs, hyperarcs, and histograms can be clicked on in this way; all cause a textual dialog window to pop-up (Figure 4).

This pop-up window contains a list of the dynamic entities that were associated with the widget of interest. For example, the pop-up for an arc contains a list of all the calls between the boxes connected by that arc; the pop-up for the allocation/deallocation histogram of the animated view gives a list of the objects that mapped to that box, when they were created, how old they were when garbage collected, and what method caused them to be created. Selection of an entry within these pop-up windows could be used to automatically position the view in a textual code browser in a future version of the tool.

3 CONSTRUCTING THE VISUALIZATION

A software engineer employs a four-stage process when using our visualization technique (Figure 5).

1. Data is collected from the execution of the system being studied, and is stored to disk.

2. The software engineer designs a high-level model of the system as a set of abstract entities that are selected to investigate the structural aspects of interest. For example, in Figure 5, “utilities” and “database” are specified as entities.

3. The engineer describes a mapping between the raw dynamic information collected and the abstract entities. Figure 5, for instance, shows that any dynamic information whose identifier begins with “foo” (such as objects whose class name starts with “foo”) is to be mapped to the “utilities” entity. This mapping is applied to the raw information collected by the tool, producing an abstract visualization of the dynamic data.

4. The software engineer interacts with the visualization, and interprets it to investigate the dynamic behaviour of the system.

The process is deliberately divided into multiple stages to increase its usability. Rather than having to complete the entire process every time any change is required for the task of discovery, iteration can occur over any suffix of the process. For example, the software engineer might begin with any extremely coarse view of the program, knowing very little about its performance; after interacting with the resulting visualization and gaining a partial understanding of the studied system’s operation, the software engineer need only alter the high-level model and corresponding mapping to generate a new visualization—there is no need to re-collect the identical dynamic information.

This process is based on the work of Murphy *et al.* [12]. We compare our visualization technique to this previous work in Section 6.

3.1 Stage 1: Gathering Dynamic Information

Dynamic system information is collected for every method call, object creation, and object deletion performed by the system being investigated. In other words, trace information is collected. This information currently consists of an ordered list of:

- the class of the object calling the method or having the object created, and

- the class of the object and method being called or returning from, or the class of object being created or deleted.

Since the tool currently uses complete trace information, the complete call stack for any given moment can be reconstructed.

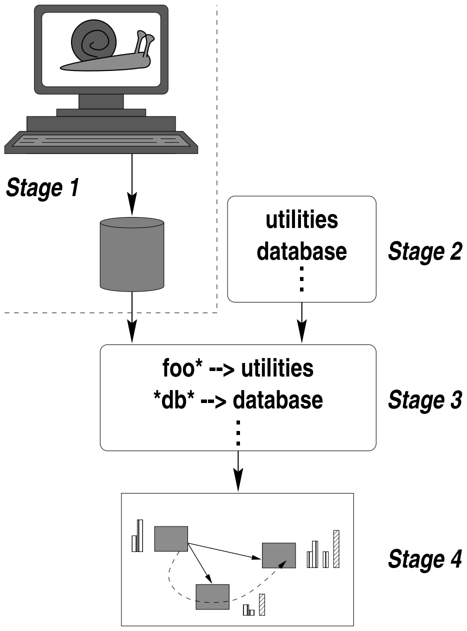

Figure 5: The process.

Because our implementation is in Smalltalk, this information is collected by instrumenting the Smalltalk virtual machine (VM) to log these events as they occur. There is nothing inherent in the tool in its use of Smalltalk; it could be as easily implemented in any language in which the execution was instrumented to collect the required information.

Because a software engineer often needs to understand dynamic problems that only occur after significant initialization of the studied system, the collection of the trace information needs to be performed only during portions of the execution. This eliminates extraneous information not of interest, and speeds up the process of collection. In our implementation, VM methods are available to dynamically activate and deactivate tracing. We used these methods to collect data for Figures 1 through 4 that included only the main iteration loop of the algorithm, excluding execution pertaining to initialization and the output of results.

3.2 Stage 2: Choosing a High-level View

The software engineer typically begins an investigation with some idea of the static structure of the system being studied. Even when this is not the case, the naming conventions and organization of the source code itself often allow some guess as to the system’s structure.

The engineer chooses a high-level structural view to use as the basis for visualization by stating the names of the abstract entities. These entities may correspond to actual system components, be aggregates or subdivisions thereof, or have little connection to reality. In Figures 1 through 4, for instance, the investigator chose two entities representing specific classes in the program (*Clustering* and *SimFunc*), and two entities representing collections of classes (*Rest* and *ModulesAndSuch*).

3.3 Stage 3: Specifying a Mapping

For the tool to indicate the dynamic interactions between the abstract entities, it needs to have a map relating dynamic system entities to the abstract ones. A map indicates that specific system entities, such as objects of a given class or methods matching a particular pattern, are to be represented by a specific abstract entity (and thus, by a box in the visualization). An engineer states this mapping using a declarative mapping language. To be usable, a mapping language must allow an engineer to easily express the relationships between entities.

The mapping language's constructs are based on the standard Smalltalk notion of structural hierarchy: methods are grouped into classes, classes into categories, categories into subapplications, and subapplications into applications. A map consists of an ordered set of entries, each of which has three parts:

1. a name indicating the level of the Smalltalk structural hierarchy being mapped,
2. a regular expression indicating the set of names to map for the particular structural hierarchy level being mapped, and
3. the name of the abstract entity to which these dynamic system entities are to be mapped.

Methods are provided for mapping a class regular expression plus method regular expression simultaneously, and subapplication, class, category, and method simultaneously. For example, say the engineer has defined an abstract entity named `foo`, and every message foo passed to classes named `*bar*` within the subapplications `dog*` should be mapped there; this would be indicated by a map entry:

```
matchingSubApplication: 'dog*'
class: '*bar*' category: '*'
method: 'foo' mapTo: 'foo'.
```

The example in Figures 1 through 4 used the following map:

```
matchingClass: 'ArchClusteringAnalysis'
mapTo: 'Clustering'

matchingClass: 'ArchModuleGroup'
mapTo: 'ModulesAndSuch'

matchingClass: 'ArchProcedure'
mapTo: 'ModulesAndSuch'

matchingClass: 'ArchSymbol'
mapTo: 'ModulesAndSuch'

matchingClass: 'ArchSimFunc'
mapTo: 'SimFunc'

matchingSubApplication: 'Schwanke*'
mapTo: 'Rest'
```

Because we are interested in visualizing the interactions between system components, the tool takes note both of the method being called and the method being executed when it was called. The same set of map entries is used to map both; the visualization itself will differentiate between incoming and outgoing calls.

Individual objects are also mapped in this way. Because it is often important *where* an object was created, we track objects not simply based on their class, but also in terms of the call stack that was present when it was created. Such an object will typically be mapped to a particular abstract entity through the mechanism described above; the object is treated as belonging to that abstract entity and is represented through its visualization (i.e., through its representation as a box).

The mapping possesses two important properties: it is *partial* and it is *ordered*. The ordering means that each system entity is mapped to a single abstract entity, the first one for which the map entry is a valid match. The mapping is partial because a software engineer does not need to express the structure of the entire system before investigating it. If a system entity fails to match every entry in the map, it is not represented in the resulting visualization. This feature both decreases the overhead for the tool and removes unwanted information from the visualization. If the engineer wants every dynamic entity to appear in the visualization, a final entry in the map of the form:

```
matchingAnything: '*' mapTo: 'default'
```

will act as a default abstract entity for all dynamic entities that "fall through" the other map entries.

4 EVALUATION

Three fundamental questions that must be answered about any software visualization are:

- Is the technique useful to software engineers trying to perform a task on a system?
- Is the technique usable by software engineers?
- For what kinds of software engineering tasks is the visualization helpful?

Evaluating a technique against each of these questions requires a number of careful, in-depth studies. These studies are warranted only after an initial determination of the coarse-grained utility of a technique. In this paper, we report on results from our preliminary investigations into the utility of our visualization technique.

In our preliminary investigations, we chose to fix the kind of software engineering task studied to be performance tuning. This task was chosen because it is heavily reliant on dynamic system information and because it tends to be delegated to "expert" developers. A visualization technique that can aid a non-expert developer in tackling performance problems would thus be beneficial in increasing the use of dynamic system information by engineers, which was one of our initial goals.

We also chose to focus on the usefulness of the technique, rather than its usability. This decision was reasonable because the main features of the technique affecting its usability have been investigated in other related domains. The iterative selection of the high-level entities and designation of the mapping by the software, for instance, are also characteristic of the software reflexion model approach from which this visualization technique is derived. Users of the software reflexion model approach have not had difficulties performing these steps [10, 11].

Our preliminary studies, then, focus on investigating the usefulness of the visualization. We report on two case studies. The first case study (Section 4.1) discusses the use of the visualization technique by one of the authors to determine why a Smalltalk implementation of a reverse engineering algorithm [13] was running slower than expected. In this scenario, we focus on the differences in information provided by the visualization technique compared to a profiler. In the second case study (Section 4.2), we had both an expert and a non-expert Smalltalk developer use the visualization to attempt to discover the cause of a performance problem with the visualization technique itself. We report on both qualitative and quantitative data collected about the use of the visualization.

4.1 Case Study #1

A hierarchical agglomerative reverse engineering algorithm attempts to automatically cluster entities, such as procedures in a C program, comprising a software system into subsystems (modules) based on a similarity function. One of the authors wanted to determine why a Smalltalk implementation of a particular algorithm [13] executed significantly more slowly than a C++ implementation.

The algorithm starts by placing each procedure in a separate module. It then iteratively computes the similarity function between each possible pair of modules; in each iteration, the most similar pair of modules is combined. The algorithm terminates when a specified number of modules are left or when no modules are similar enough to be combined.

The performance investigator had knowledge of the design of each program, but had not implemented either program. To examine the performance of the Smalltalk implementation, the investigator first used the IBM VisualAge for Smalltalk execution profiler. With this tool, a user can either sample or trace the execution of an application, and then view collected statistics, such as the amount of execution time spent in particular methods or the number of garbage collection scavenges. After perusing several of these views, the investigator determined about 16% of the execution time was spent in methods of the `ArchClusteringAnalysis` class that contains the main iteration loop, 5.5% was spent in methods of the `ArchCache` class that acts as a cache for already computed similarity values, and 4.6% was spent in computing new similarity values. This result was not surprising. The information confirmed the investigator's understanding of how the program works, but did not provide any hints as to whether the performance could be enhanced.

The investigator next applied the visualization technique, choosing a high-level model consisting of four entities. One entity, *Clustering*, represented the `ArchClusteringAnalysis` class. Another, *SimFunc*, represented the class that had methods for computing the similarity function. A third, *ModulesAndSuch*, represented the functions and modules whose similarity was to be compared, and a fourth, *Rest*, represented all other classes comprising the program. The mapping associated the appropriate classes (and sub-applications) with these boxes. The investigator collected trace information for the main iteration loop of the program and then began interacting with the visualization.

Playing through the abstracted information, the investigator noted the large number of objects (over 4500) associated with the *SimFunc* entity. The investigator viewed the summary and queried it for the objects associated with *SimFunc*'s box. The object list contained many `Set` and `MethodContext` objects (Figure 4). These results confirmed that the cost of computing the similarity between two modules was high and should be minimized. Returning to a "play" through the visualization, the investigator noted that the ratio of calls from *Clustering* to *Rest* and from *Rest* to *SimFunc* was lower than expected. Prior investigation had shown that the majority of the calls between *Clustering* and *Rest* were due to calls on the `ArchCache` object; calls from *Rest* to *SimFunc* represent new computations of similarity.[1] This insight led the investigator to study the `ArchCache` class. The investigator found that the "key" value used to store and access similarity values in the cache was not causing as many hits as it could. A slight modification to the formation of keys resulted in an increase of just over 25% in the speed of the program.

The visualization technique aided this performance-tuning task by presenting information that caused the investigator to ask, and answer, the "right" questions about the implementation. Insight into structural interactions in the system helped the investigator narrow in on the algorithmic problem. The investigator made use of the both the interaction and object allocation and deallocation information, the summary view, and the ability to play, and re-play, through the traced execution.

4.2 Case Study #2

In the second case study, the tool was used to investigate its own performance problems; specifically, due to a structural design flaw, it was faster to step forward than to step backward in the visualization tool. This flaw centered on the fact that the implementor had chosen to generate cels on the fly and often used simple linked lists to hold the required information for the arc annotations; as a result, adding to these lists via the method

[1] A better design for the program would have been to hide the cache behind the `ArchSimFunc` interface.

```
addInteractionsFrom:to:between:
```

was fast, but removing from the lists via

```
removeInteractionsFrom:to:between:
```

required a linear-order search through each list. The implementor of the tool had discovered this flaw and informed the experimenters of its existence and its cause.

To prepare for the studies, the experimenters gathered a trace consisting of stepping forwards and backwards in the visualization tool.[2] An initial high-level model and mapping were also prepared for the participants as the short study periods were intended to focus on the visualization itself, rather than the process of creating a visualization. The high-level model was very simple, and can be seen in the visualization shown in Figure 6; the classes used by the tool all had names that began with a two- or three-letter prefix, and thus were mapped to abstract entities with these prefixes as names.

In a separate session each, a previously collected trace was given to two experimental participants: an expert at solving performance problems in Smalltalk applications, and a non-expert in solving performance problems in any language. Each participant was given an introduction to the tool and a short training session in which each had the opportunity to use the tool on a toy problem. Then, the symptom of the flaw in the tool was explained, and the parameters and interaction that we had traced were described. Each was asked to determine three or fewer points of interest within the source code for the tool that they saw as being good candidates for more detailed analysis; they were also asked to answer a set of questions periodically in regards to their perceptions of the tool and progress in their task. We audio-taped these question and answer sessions. We also captured automatically a log of the participants' navigation pattern through the visualization using instrumentation built into the prototype.

4.2.1 The Expert Participant

The expert participant began with a ten-minute inspection of the summary view: the *Gp* box was seen to have the most objects allocated, and most of these were immediately deallocated. Querying the attendant Allocation Pattern histogram showed that many of these objects were of the classes `Point`, `MethodContext`, and `BlockContext`.

The animated view was then used, both in step forward and backward mode and in play mode, to examine the range of cels where many of these objects were being allocated; a repetitive call pattern was observed between the *Gp* and *Cdf* boxes. The arcs and hyperarcs between these boxes were queried for details, and the methods involved in this pattern were found by the participant. A separate code browser was then used to investigate the details causing this behaviour. After studying the system for an hour, the participant decided that the likeliest cause was in the methods

- `removeInteractionsFrom:to:between:`, and
- `addInteractionsFrom:to:between:`.

The participant noted the similarity of code in these two methods. This observation made sense because the fundamental problem was due to the data structure. The participant was thus able to indicate a useful point to continue the investigation, as had been requested at the start of the study.

The expert participant liked two features of the tool in particular:

- the summary view, although the participant stated: "in this case [the effect] was slightly obvious [in the summary view]—it may not be so obvious in other cases"; and
- the animation of the hyperarc resulting from pressing "play", because of the way one can watch "how things go into loops or circles or watch the communication back and forth between different things, or specific things."

The expert participant felt the tool lacked two desirable features:

- integration between it and a traditional code browser, so one could, for example, select a method in a pop-up detail window and have the code browser display that method; and
- the lack of ability to view a detailed stack dump, comparable to that available from a Smalltalk debugger, particularly so that the parameter types being passed could be seen (this cannot be seen from the static code because Smalltalk is dynamically typed). The actual values being passed were deemed desired in some instances.

Code browser integration is a desired feature that has not yet been implemented; the tool has been designed to accommodate this change. The tool did allow the participant to narrow the search to particular points of interest that could then be investigated via a debugger or similar means. The desire for greater, integrated information from the tool is understandable, but runs contrary to its design philosophy of complementing existing techniques—it is not intended to supercede the use of a debugger. This desire also highlights the tension between off-line and on-line approaches to accessing dynamic information.

4.2.2 The Non-Expert Participant

The non-expert participant made extensive use of both the object histograms and the allocation/deallocation bars in the detailed view to investigate the performance problem. Specifically, the participant would find cels in which object deallocation was not keeping pace with object allocation (i.e.,

[2] The visualization tool had to be run on a different, pre-existing execution trace. A toy example was used for this purpose, but choice of input was not a factor in the tool's symptoms. The second participant actually received a trace of only a step backwards.

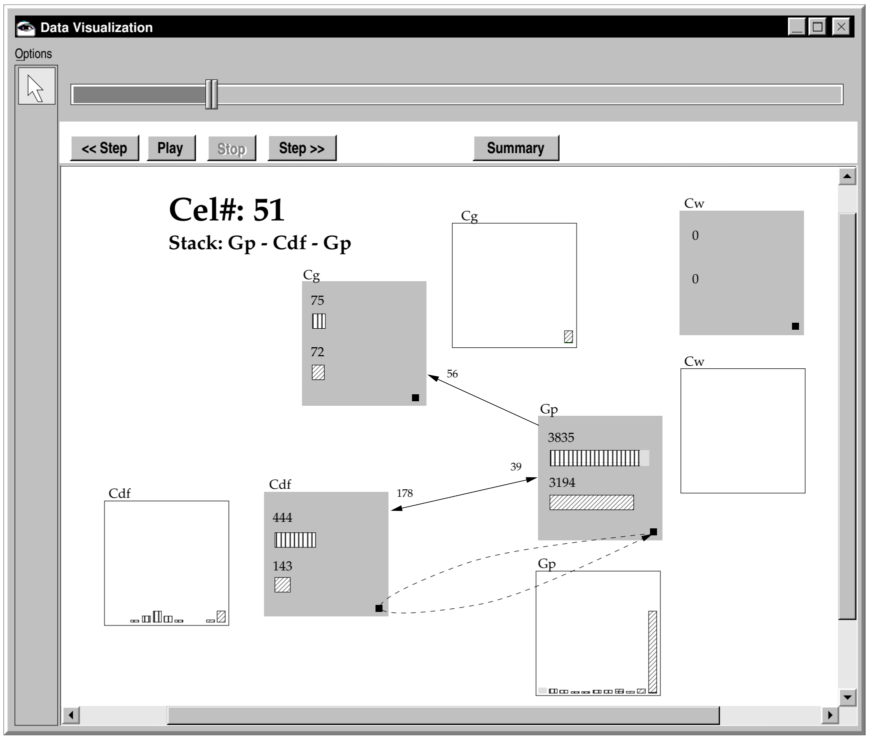

Figure 6: Case study #2 visualization.

the green bar—shown herein via a vertical-line pattern—was longer than the red bar—shown herein via a diagonal-line pattern—within a box) and would then step forward to see when objects were being allocated. Queries on the associated histograms were then used to determine the classes of the allocated objects. Less frequently, the participant would investigate the calls involved with the allocations.

For the first forty minutes, the participant worked solely with the visualization tool. After that, the participant began to use the Smalltalk code browsers to study the associated code. After approximately an hour with the tool, the participant had identified two methods, including the `removeInteractionsFrom` method, as a point in the code at which to continue the investigation. This determination was based, in part, on noticing a correlation between an increase in message sends between the *Gp* and *Cg* boxes and the number of objects allocated by *Cg*. Similar to the case of the expert participant then, the non-expert found the correct area of code to investigate, which was the task that had been posed.

The non-expert found the deallocation age histograms and the ability to determine the correlation between abstract information to method and object names by clicking on histograms and interactions in the visualization particularly helpful. However, the non-expert indicated a desire for different displays of this information, finding the "screen with all the methods [was] too cluttered." Similar to the expert, the non-expert desired more integration with other Smalltalk tools, such as the code browser. For instance, the participant wanted to be able to select a call from a list of interactions and visit that call site in the code.

During an interview part of the way through the study period, the participant noted that it was difficult to attack the task because of a lack of knowledge of what could cause performance problems. The visualization tool provided some clue as to how to proceed because of its emphasis on particular dynamic information. The applicability of the dynamic information chosen for other tasks requires further research.

5 DISCUSSION

Key features of our technique include off-line operation, a navigable visualization of the collected data, cels based on a running summary, and the use of a declarative mapping to abstract fine-grained information about a system's execution. We discuss each of these features and our use of trace information.

5.1 Off-line Operation

Using an on-line visualization technique can be a slow, unidirectional procedure. Taking the technique off-line and separating the visualization from the system execution can achieve two benefits.

First, it allows the information to be preprocessed as a whole prior to visualization, enabling the generation of summary information about the entire execution. For the performance tuning tasks described in the case studies, summary information was used to provide clues about which parts of the system to investigate as potential sources of the problem. After accessing summary information, the users returned to investigate detailed parts of the execution.

Second, it allows any partial trace of an execution to be reviewed without having to re-run the entire execution. This review capability permits the visualization to be navigable in a way that is not possible for an on-line technique. Not only may the trace be replayed from any arbitrary point, but also it may be played backwards, or at a rate that is independent of the speed of the original execution of the system being studied.

5.2 Navigable Visualization

One advantage of an off-line visualization approach is the navigation capability provided to the software engineer. The user can unfold the execution in a forward, "play", mode, but then can perform detailed investigations of particular parts of the execution by moving the visualization both forward and backward. In our current prototype, we do not associate any information about the actual execution time with the off-line navigation. Each step forward or backward in our visualization takes time proportional to the display time of the next cel, rather than representing the length of time required by an associated method call, allocation or garbage collection. For some tasks, including performance tuning, it would sometimes be helpful to have steps between cels represent the system running time.

5.3 Running Summary

We believe that separately displaying individual events, or small groups of contiguous events, makes for an insufficient visualization of a system execution because of a lack of connection to the greater context of that execution. Some sort of summary information is also needed.

We considered two means of providing such summary information: a single summary picture, such as that in Figure 3, and a set of pictures showing the change to the state of the system over individual intervals of its execution ("delta" information), which is not provided by our tool. But neither alone would be sufficient to illustrate the dynamic nature of the information we are attempting to visualize. The summary picture clearly does not contain any temporal ordering of events—it is difficult to look at one and mentally reconstruct the sequence of events that produced it. Furthermore, this summary alone cannot contain enough detail about the execution to be useful without becoming so cluttered that it is rendered unusable. Delta pictures address the concern of visualization of the temporal nature of the information; however, it is difficult to understand the relationship between a delta picture and the execution *in toto*. To reach a compromise between these alternatives, we chose to provide a running summary of the execution within the individual cels. This implicitly provides the temporal component of the summary information while maintaining context for the delta information within a cel.

Two other alternatives to maintain context are possible. In the first, we could begin with a summary view such as that provided by our tool. But rather than being a single, static picture, it could also be divided into a sequence of cels each of which would show the same summary information while highlighting in a different colour, say, the information that was changed or added over the represented interval, such as the directed arcs that were traversed, or the subset of objects that were deallocated. The second alternative is similar, but instead of highlighting only the information that is different for that interval, a running summary of all the information that had changed from the start of execution of the system to the current interval would be highlighted. Both can suffer from the fact that a complete summary view can quickly become too detailed, leading to information overload. However, both these schemes could be used to complement the delta plus running-summary combination currently used in our cels; we have not yet investigated this possibility.

5.4 Mapping Objects

Each cel maps objects to abstraction units. Associating an object with an abstraction unit using our declarative mapping approach requires a means of "naming" objects. We chose to name—more precisely, identify—an object based on where it is created in the code: a software engineer identifies objects mapping to a particular abstraction unit by describing a part of the call stack that exists when one of the objects is created. This approach has the advantage that an engineer can identify collections of objects by perusing the source code and describing the locations where relevant allocations occur. Another possible choice would be to name objects based on their class. However, this approach to naming would not allow objects of the same class to be mapped to different abstraction units, limiting the ability of the engineer to differentiate distinct uses of classes.

Currently, the mapping provided by the engineer is applied uniformly to all dynamic information collected as the system executes. A ramification of this decision is that once an object is associated with an abstraction unit, it remains associated with that unit for the duration of the visualization. Sometimes, though, it may be useful to modify the association of objects to abstraction units over the course of the execution. For instance, if an object is created in one subsystem, but is then immediately passed as an argument to another subsystem, it may be useful to capture the "migration" of the object. Supporting this migration would require not only a means to allow the engineer to describe when and how the migration would occur, but also would require updates to the use of histograms for object allocation and deallocation. Further understanding of how this capability might help in the performance of tasks is required before support is added.

5.5 Dynamic Information

Our current prototype visualizes trace information collected about a system's execution. Trace information has the benefit that it is complete: all object interactions, allocations, and deallocations are included in the trace. Complete information is easy for the engineer to reason about. However, trace information has the often cited problem of being voluminous [9, 2, 8]. Tracing even small pieces of a system's execution can result in a huge amounts of data. Although we have been able to successfully use trace data to investigate some performance problems, the use of trace information limits the flexibility and usability of our current prototype. We plan to investigate the use of sampled information as a basis for our prototype to overcome some of these limitations.

6 RELATED WORK

De Pauw *et al.* have developed a number of visualizations to describe the execution of an object-oriented system, including inter-class call cluster diagrams, inter-class call matrices, a histogram of instances, and an allocation matrix [1]. All of these visualizations show fine-grained execution information about individual classes and objects. The utility of these visualizations degrades as the size, measured in the number of classes, of a system grows. Several other similar object- and class-level visualization approaches have been developed (e.g., [6, 5]); these techniques share the same scalability problem.

Lange and Nakamura in the Program Explorer tool allow the developer to integrate, off-line, static and dynamic information about a program to aid comprehension activities [7, 8]. For instance, they show how this combination of information can help a developer find and investigate instances of design patterns in a system. The visualizations they produce are also at a fine-grained level. Vlissides *et al.* use a different notion of pattern, which they refer to as execution patterns, to help developers investigate the large amount of fine-grained execution information available about a system [3]. Specifically, they allow a developer to query an on-line animation for patterns appearing in a dynamic execution stream. In both the Program Explorer and execution pattern approaches, the developer must apply detailed knowledge about a system to formulate appropriate queries.

Jerding *et al.* have applied the information mural approach to create a scalable visualization of fine-grained program events [4]. The result, an execution mural, places classes vertically on the screen and uses single pixel vertical bars, with various colouring approaches, to indicate calls between classes. The interactions occurring in the system are then shown across the screen. Using this approach, thousands of interactions occurring between objects can be visualized on one screen. The authors extend these ideas to a Pattern Mural that provides an information mural display of automatically detected common occurring sequences of calls (patterns) in the execution. Although this approach may help a developer find unexpected patterns, or verify existing patterns in the code, it still visualizes only fine-grained information about the system.

The approach taken by Sefika *et al.* differs in allowing a developer to utilize coarse-grained system information to produce visualizations [14]. Using their technique, a developer may introduce various abstractions into the system instrumentation process, including subsystem, framework and pattern-level abstractions. The abstractions can then be used as a basis for several visualizations including affinity and ternary diagrams. The coarser-grained visualizations produced with this technique make it easier for developers to investigate inter-component interactions in large systems than previous approaches.

Some of the design decisions Sefika *et al.* made in developing their technique limit its flexibility. Choosing an on-line approach permits a link between the speed shown in the visualization and the execution speed. However, as we have discussed, an on-line approach limits the modes of investigation available to an engineer. Choosing an approach that hard-wires the abstractions of interest into the instrumentation process provides an effective data gathering mechanism; however, it decreases the usability of the technique by making it more difficult for an engineer to apply it to a new system. We have been able to easily apply our technique to different systems because of the separation in our process between data gathering and visualization.

Our visualization technique builds on the software reflexion model technique developed by Murphy *et al.* [12, 10]. The reflexion model technique helps an engineer access both static and dynamic information about a system by enabling a comparison between a posited high-level model and a model representing information extracted from either the static source or from a system's execution. Similar to our visualization technique, the software reflexion model depends on a declarative mapping language. Our visualization technique extends the reflexion model work in three fundamental ways: by applying the abstraction approach across discrete intervals of the execution with animation controls, by providing support to map dynamic entities rather than only static entities, and by mapping memory aspects of an execution in addition to interactions. Our visualization technique also uses the running summary model rather than the complete summary model used in the reflexion model approach.

7 SUMMARY AND FUTURE WORK

Condensing dynamic information collected during a system's execution in terms of abstractions that represent coarse system structure, such as frameworks and subsystems, can help software engineers investigate the behaviour of a system. We have developed a visualization technique that allows engineers to flexibly define the coarse structure of interest, and to flexibly navigate through the resulting abstracted views of the system's execution. Our approach complements and extends existing visualization techniques.

Our preliminary investigations into the usefulness and usability of the visualization indicate it shows promise for enhancing a software engineer's ability to utilize dynamic information when performing tasks on a system. To date, we have focused on the use of dynamic information to aid one particular software engineering task—performance tuning. We intend to continue our investigations into the utility of the entire technique through more extensive case studies on a wider range of tasks on larger systems. Although there is evidence elsewhere [10, 11] that the iterative mapping approach is usable for static information, our further studies will investigate if this remains true for dynamic information.

ACKNOWLEDGMENTS

This work was funded by a British Columbia Advanced Systems Institute Industrial Partnership Program grant, by OTI, Inc., and by an NSERC research grant. We thank Edith Law for participating in the case study, and we thank the anonymous reviewers for their comments.

REFERENCES

[1] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the 1993 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '93)*, pp. 326–337, 1993.

[2] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In *Proceedings of the 8th European Conference on Object-oriented Programming (ECOOP '94)*, pp. 163–182, 1994.

[3] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, pp. 219–234, 1998.

[4] D. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proceedings of the 19th International Conference on Software Engineering*, pp. 360–370, 1997.

[5] K. Koskimies and H. Mössenböck. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. In *Proceedings of the 18th International Conference on Software Engineering*, pp. 366–375, 1996.

[6] C. Laffra and A. Malhotra. Hotwire—a visual debugger for C++. In *Proceedings of the USENIX C++ Technical Conference*, pp. 109–122, 1994.

[7] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of the 1995 ACM Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '95)*, pp. 342–357, 1995.

[8] D. B. Lange and Y. Nakamura. Object-oriented program tracing and visualization. *Computer*, 30(5):63–70, May 1997.

[9] J. R. Larus. Efficient program tracing. *Computer*, 26(5):52–61, May 1993.

[10] G. C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. Ph.D. Dissertation, Department of Computer Science and Engineering, University of Washington, 1996.

[11] G. C. Murphy and D. Notkin. Reengineering with reflexion models: A case study. *Computer*, 30(8):29–36, August 1997.

[12] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 18–28, 1995.

[13] R. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pp. 83–92, 1991.

[14] M. Sefika, A. Sane, and R. H. Campbell. Architecture-oriented visualization. In *Proceedings of the 1996 ACM Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*, pp. 389–405, 1996.