Execution Patterns in Object-Oriented Visualization

Wim De Pauw, David Lorenz*, John Vlissides, and Mark Wegman

IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598 USA
{wim,lorenz,vlis,wegman}@watson.ibm.com

Abstract

Execution patterns are a new metaphor for visualizing execution traces of object-oriented programs. We present an execution pattern view that lets a programmer visualize and explore a program's execution at varied levels of abstraction. The view employs visual, navigational, and analytical techniques that accommodate lengthy, real-world traces. By classifying repetitive behavior automatically into high-order execution patterns, we drastically reduce the information a programmer must assimilate, with little loss of insight.

1 Introduction

The gap between a program's static specification and its dynamic behavior is particularly large in object-oriented programs. Experienced developers migrating from procedural to object technology often complain that they can't discern flow of control in object-oriented programs. Their complaints grow louder when they deal with one or more frameworks. The inversion of control from which a framework derives its leverage [22] nonetheless gives developers feelings of insecurity, because their understanding of a program is bound tightly to knowledge of control flow. When they can't trace the program's logic, they can't predict its behavior, and so they can't change what it does—rendering the framework useless.

Object-oriented program visualization systems such as Ovation [9, 10] and Program Explorer [18] have sought to bridge this gap. All work in much the same way. They instrument the code to be visualized so that it produces a trace as a side-effect of its execution. The trace contains far too much detail (e.g., every method invocation) to understand as is. Instead, the visualization system interprets the trace and generates one or more animated renditions of the program's execution. The renditions use a variety of notations and cognitive devices to convey information succinctly and comprehensibly. The visualization can help humans make sense of millions of method invocations on hundreds of thousands of objects. But to be effective, the visualization system must consolidate and present just the right information on the screen, in just the right way. There are many ways to consolidate information, and their effectiveness varies tremendously.

To understand how to present information effectively, consider how traces become large in the first place. One reason is sheer complexity. A nontrivial object-oriented program may enlist hundreds or even thousands of classes to do its job. One would expect large traces from such programs.

Yet a program doesn't have to be statically complex to yield large traces; even a small program may produce them if it executes the same code many times. Programming languages let us specify repeated executions of (almost) the same scenario through recursion, functional decomposition, template functions, and a variety of control structures. These remarkably compact specifications produce patterns of similar if not identical instructions as the program executes, greatly lengthening the trace but adding little to its information content. Hence even a simple program can yield huge, incomprehensible traces.

Nevertheless, trace data can be comprehended (1) if it can be summarized into succinct, abstract nuggets that lack extraneous detail (details are provided on demand, not unsolicited), and (2) if similar patterns in the trace can be condensed into a smaller number of more general patterns that recur at specified frequencies.

To this end we introduce a variation of Jacobson's interaction diagrams [16] for presenting patterns of execution—an **execution pattern view**—that accomplishes two aims:

1. It lets a programmer observe any part of the program's execution at various levels of detail. The programmer avoids being overwhelmed by execution information through careful, se-

*David Lorenz is currently at Technion, Israel Institute of Technology (david@cs.technion.ac.il).

lective control of what is divulged, with detail presented on demand.

2. It detects and presents generalized patterns of execution in which one pattern subsumes many parts of the trace.

In the next section we describe the elements of the pattern view. We use a scenario to motivate the mechanisms for navigation and detail elaboration. Then we discuss generalizations of execution patterns. We give criteria by which patterns are automatically considered similar and explain our choice of color and shape to best convey the information. Finally, we compare our execution pattern view to related work and offer concluding remarks.

2 Execution Pattern Notation

Jacobson's interaction diagrams are popular because they depict dynamic behavior clearly and compactly. Figure 1(a) shows a simple interaction diagram. Vertical lines, or **rails**, represent objects. Arrowheaded lines represent message sends: there is an arrowheaded line between object *A* and object *B* for each message *A* sends to *B*, with lines for later messages appearing below those for earlier ones. Thus time progresses downward. All method invocations on an object appear as rectangles superimposed on the corresponding rail.

An interaction diagram's layout emphasizes the program's thread of execution. Before interaction diagrams it was common to depict this information as directed graphs, with nodes representing objects and arcs representing message sends. The problem was scalability. It was difficult to depict more than a few message sends between objects without clutter. By representing time explicitly—that is, by mapping it to the vertical axis—interaction diagrams can depict many more message sends unambiguously.

While interaction diagrams are better than directed graphs at depicting nontrivial interactions, even these diagrams do not scale to complete program executions. Space is consumed quickly as execution time and the number of objects increase. There are ambiguities as well. The order of rails along the horizontal axis is undefined; often it is chosen to minimize line length, crossovers, or both. Nor is it easy to discern the lifetimes of recursive calls, since they are subsumed by a single rectangle. For example, it's unclear in Figure 1(a) whether the message from *A* to *D* is sent within the recursive call to *A*. One can make recursion more explicit by superimposing rectangles, but that approach quickly gets unwieldy as the recursion deepens.

The execution pattern view addresses these shortcomings by unfolding the graph into tree structures like the one shown in Figure 1(b). This layout emphasizes the progression of *time*, not the thread of control. You read the graph left to right and top to bottom, just as you would printed English. Later messages appear further to the right on the same line or further down the screen than earlier ones.

In addition to these topological improvements, we use color to indicate the class of object. Figure 2 shows the execution pattern view at an early point in a program's execution. The legend in the lower left indicates that objects of class `AA` are orange, objects of class `MM` are yellow, and objects of class `QQ` are blue. To the left of an object is the sender of the message it received; to its right, the messages it sends. Each object in the view is identified with a numerical identifier ("ID") in the upper left of its colored box. Messages are shown as labeled arrowheaded lines, the label identifying the name of the message. In the figure, the message `fA` is sent to an object of class `AA` (ID=2). The object responds to this message by sending a message `fM` to the `MM` object (ID=3) and then sending a message `fQ` to the `QQ` object (ID=4).

A tree representation simplifies things considerably. Its unidirectionality in both axes makes it easier to read than an interaction diagram (where messages may bounce across many rails in either direction). It scales better, too. Horizontal space is mapped not to objects but to the call sequence, which is uniquely defined and more tightly bound—several objects may occupy the same column. Vertical space is used more efficiently as well (compare Figures 1(a) and 1(b)).

3 Manipulation and Navigation

The execution pattern view has interactive capabilities that capitalize on the tree structure's properties. Suppose we want to understand how a typical `Bus-Observer` object in a program handles the `update` message. The user can initiate a search for execution patterns on various criteria, such as the involvement of a particular class, object, or message name. In this case, the programmer would specify the class `Bus-Observer` and possibly the

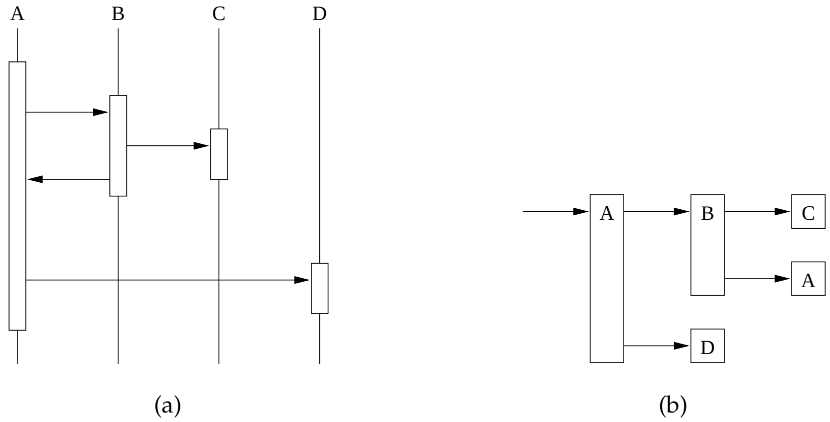

Figure 1: Simple interaction diagram (a) and its corresponding execution pattern (b)

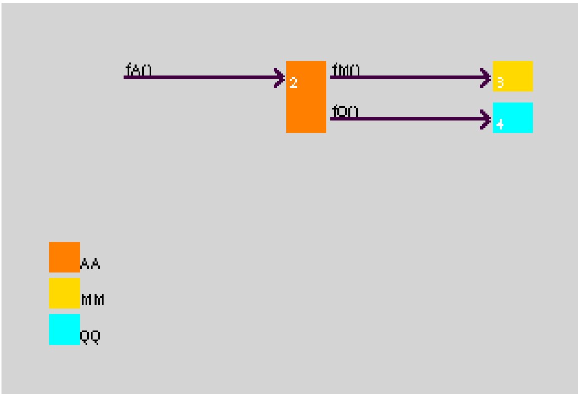

Figure 2: Simple execution pattern

`update` message. Then he can browse through views, like Figure 3, depicting the ways different `Bus-Observer` objects handled the `update` message.

In this example, an initial message `update` is sent to the black `Bus-Observer` object (ID=762). It responds by sending a message `notify_pending` to the purple `EClassModel` object (ID=761). Next, the `Bus-Observer` objects sends a `Phrase` message to the orange `Annobus` object (ID=758). Finally, it sends another message to the purple `EClassModel` object (ID=761).

3.1 Collapsing and Expanding Subtrees

Now suppose the programmer wants to explore the response of this `update` message to the `Bus-Observer` object (ID=762) in more detail than Figure 3 provides. You can see that the depiction of the `Annobus` object (ID=758) and the lower (i.e., later) depiction of the `EClassModel` object (ID=761) have a beveled border, making them look raised as opposed to flat. A raised rectangle indicates that the object reacted to the stimulus by sending one or more messages.

Clicking on a raised object reveals the message(s) that this object sent along with the object(s) that received the message(s). After clicking on subsequent raised objects, we get a view like the one in Figure 4. (Note the self-invocation of the `EClass-Model` object.) All objects appear flat now, meaning that no hidden messages remain. If we don't want to see a part of the execution, we can **collapse** part of the view by clicking on a flat object, thereby hiding its responses. The object will now appear beveled as before.

This simple technique of expanding and collapsing is a helpful navigation tool. The programmer can selectively drill down to any level of detail without being flooded with information. Moreover, the metaphor is reminiscent of how encapsulation works in object-oriented programs: details of how an object performs a given task are hidden unless sought explicitly.

3.2 Changing Context

A programmer is likely to ask two questions at this point: "Who sent the initial message `update` to `Bus-Observer` 762?" and "What was the context of that message?" The system can take us up a level to view the sender of this `update` message.

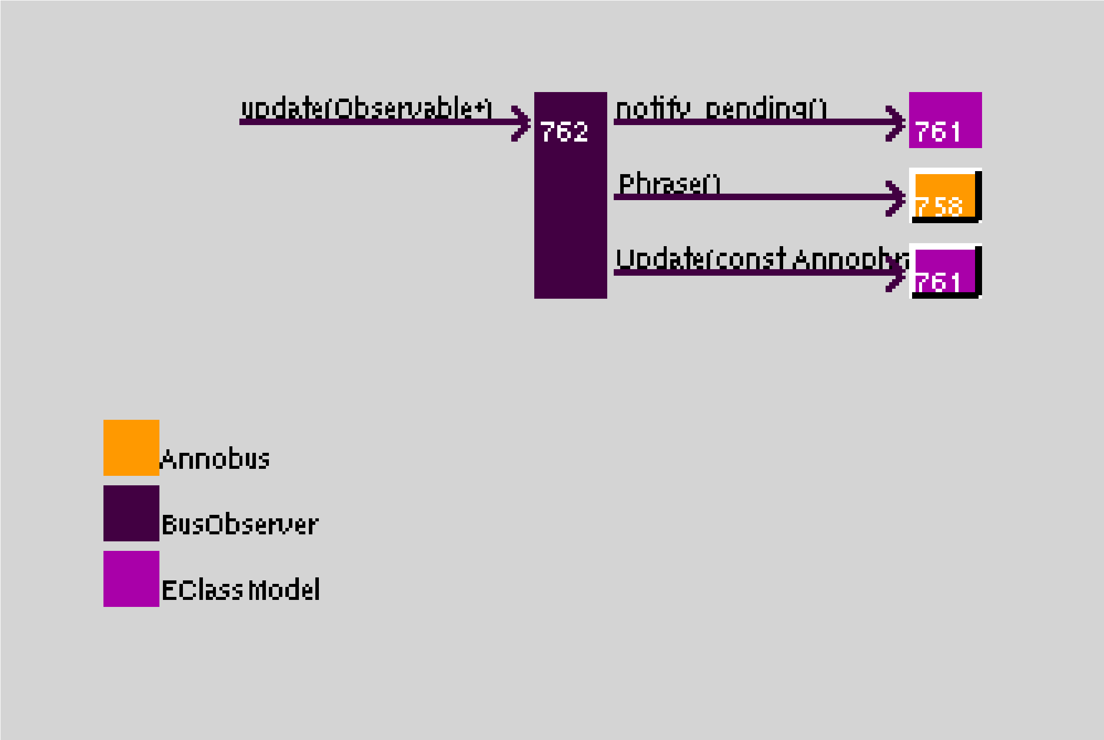

Figure 3: Message `update` sent to `Bus-Observer` object, and its response

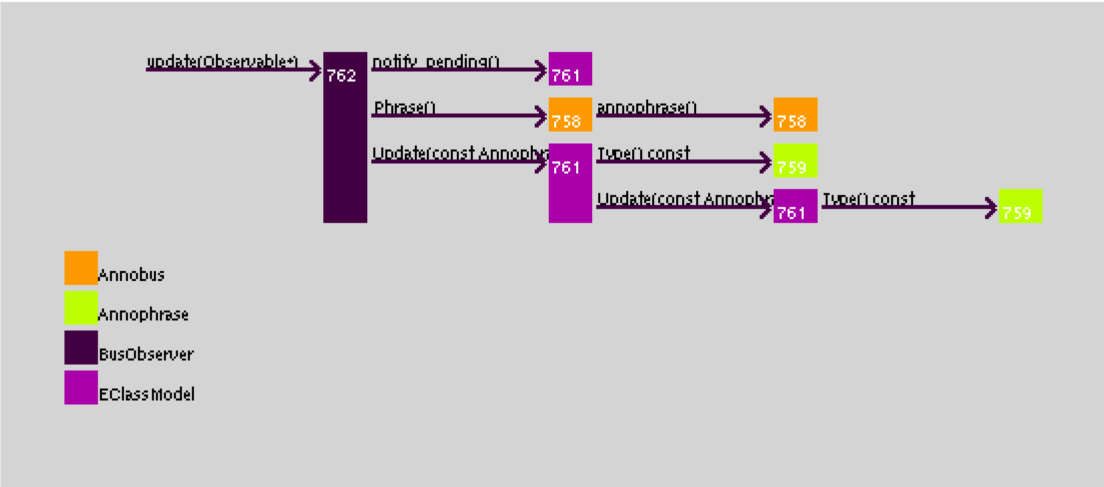

Figure 4: Message `update` sent to `Bus-Observer` object and its fully expanded response

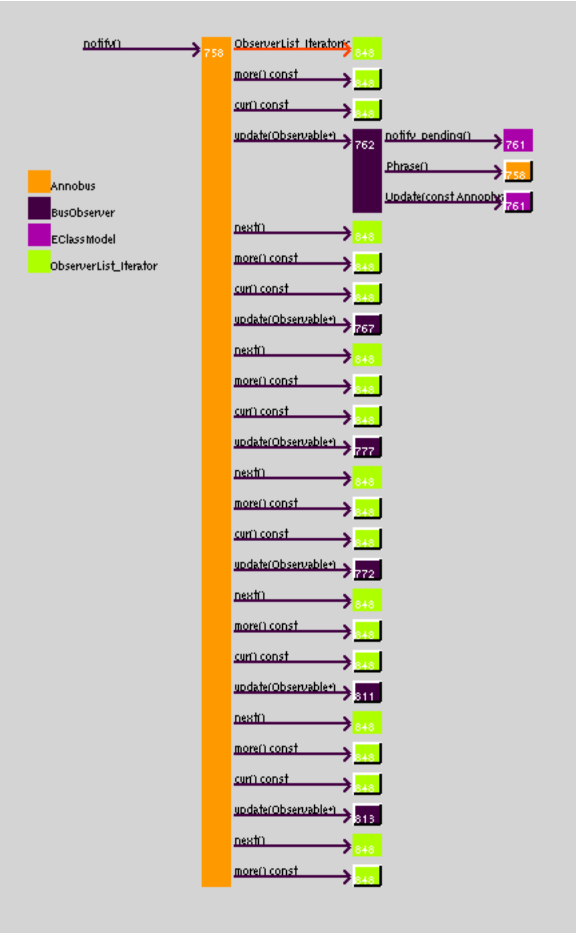

Figure 5: Revealing the sender of the `update` message: an `Annobus` object

The result is the view in Figure 5, which reveals that `Annobus` object 758 sent the `update` message to `Bus-Observer` 762. `Annobus` 758 also sent messages to several other objects in response to a `notify` message it received.

3.3 Filtered Expansion

What if the user wants to see *every* message a `Bus-Observer` receives, filtering out as many other interactions as possible? That's done by expanding only those nodes in the tree that lead to a `Bus-Observer` object, collapsing all subtrees not containing `Bus-Observer`. We call this **filtered expansion**.

Filtered expansion can be useful for identifying patterns that entail object instantiation. Object-oriented programs do nothing if they don't create objects, and understanding *how* they create them is often more important than knowing *when* they do. To track down such patterns, we can filter out all messages that do not ultimately lead to a creating (and destroying) message.

3.4 Repetition

Figure 5 shows considerable repetition in the execution pattern: a series of four patterns (initiated by `more`, `cur`, `update`, and `next`) is repeated six times. Indeed, if we were to expand the black and green colored objects, we would find that the collapsed portions exhibit the same pattern, too. While this example has a repetition factor of only six, often repetition factors are much higher. Visualizations of such repetitive sequences rarely pull their weight: they take a lot of screen space without adding much information.

The system can automatically detect repetitions in patterns; the result is shown in Figure 6. The sequence that was repeated six times now appears raised, indicating the repetition factor in its lower left corner. This compact representation corresponds to a loop in the source code.

Further, just as loops can be nested in source code, the system detects and represents loops at any level of nesting. If we navigate up a few times, we get a view that looks like Figure 7. The innermost frame indicates six repetitions, the middle frame two, and the outermost frame 31. Obviously, this pattern would take a lot more space if the repetitions were expanded. Objects within the raised frame may be expanded and collapsed as before: note how some objects in Figure 7 are flat (expanded) and some are raised (collapsed).

This example shows a repetition of an iteration sequence. Another important kind of repetition is a recursion sequence. While iterative patterns repeat in the vertical direction, recursion shows up as repetition in the horizontal direction. We can apply the same techniques to make recursion more compact.

3.5 Zooming and Panning

The system supports zooming and panning, two traditional perspective-changing mechanisms. Figure 8 shows a zoomed-out view of the highest level of the program. Although we are using the same pattern view, zooming serves other purposes than those we've discussed. At this mag-

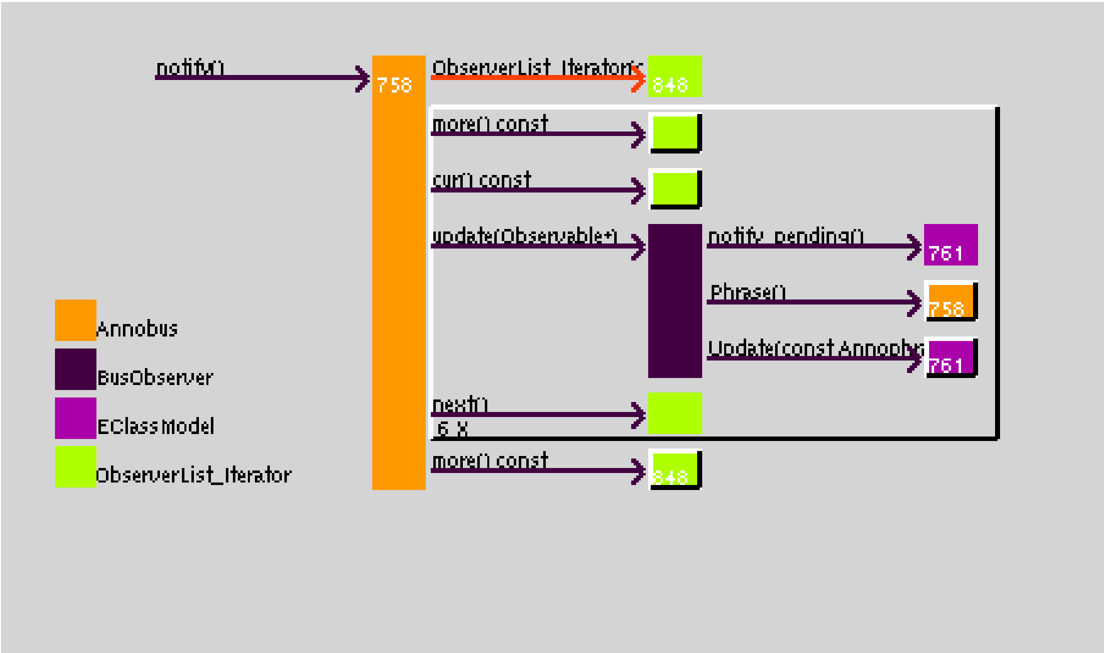

Figure 6: Raised frame indicates repetition (6X) of the sequence inside

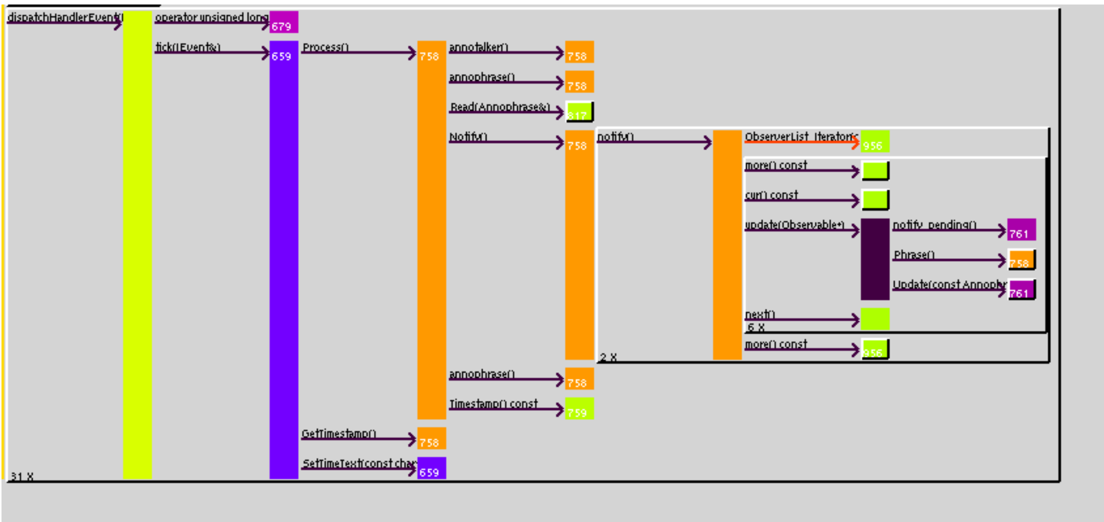

Figure 7: Three nested repetitions

nification level it's impossible to discern individual messages or objects. (In fact, the view omits text and other minutiae when they are too small to read.) What we get here is a general idea of the different phases in the program. The dominant colors indicate the classes that are prevalent in each phase. Note also that the width of the pattern reflects the stack depth at a particular phase of the program.

3.6 Flattening and Underlaying

Sometimes even the stack depth gets overwhelming. In that case you might have to ignore certain objects, classes, messages, or combinations thereof. You might decide to ignore library classes, for example, or private methods. Filtered expansion (Section 3.3) doesn't help you here. A collapsed node encapsulates the *entire* response to a message send. Once collapsed, none of the details are visible.

The execution pattern view lets a node encapsulate a specific outgoing message by collapsing *only* the receiver of the message. We call this **flattening** the receiver. Both the message and the receiver's identity are hidden; however, the fact that a receiver exists and its type are still discernible.

Figure 9 illustrates different flattening combinations. In each case only an indication of the receiver remains after flattening—just enough to reveal its existence and color (class) but not its label (identity) or the arrowheaded line (message).

Another elision mechanism removes a node from the graph even as it highlights the node's encapsulation of its children. Figure 10 shows a node **underlaying** the subtree it encapsulates. The messages sent by the underlaying node are hidden, thereby reducing visual clutter and saving horizontal space.

Collapsing, flattening, and underlaying differ in their elision properties. Collapsing achieves the greatest elision and hence the greatest space savings, but it leaves no clues about the patterns it elides. Flattening retains type information in exchange for only slightly more real estate, but it hides object identity. The space savings are least with underlaying, but it simplifies the view with minimal information loss: only a few messages are hidden.

The execution pattern view lets you mix and match these techniques to exploit their strengths. For example, a node underlaying a flattened sub-

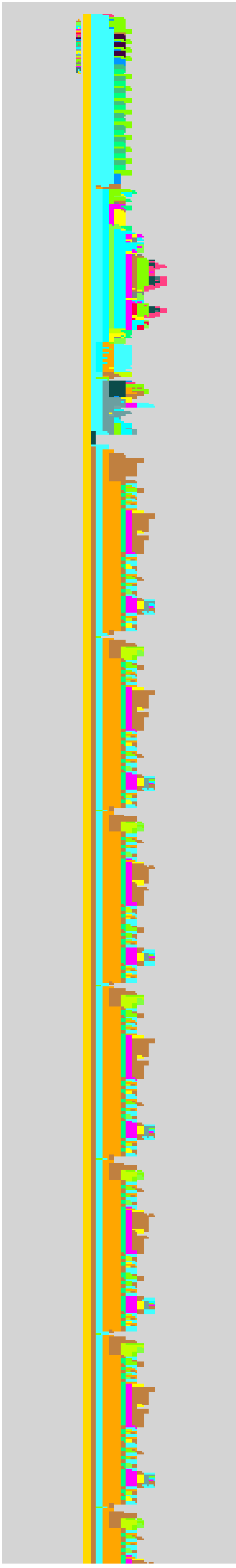

Figure 8: Zoomed-out view of the entire execution

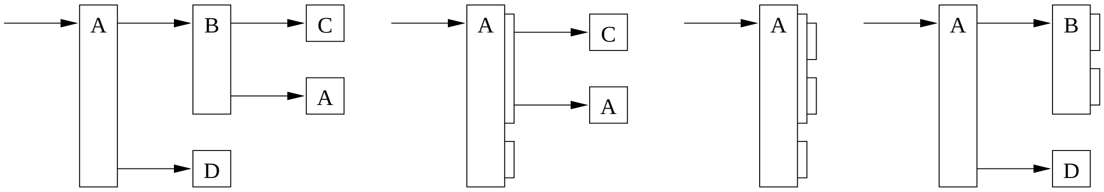

Figure 9: Schematic view of flattening

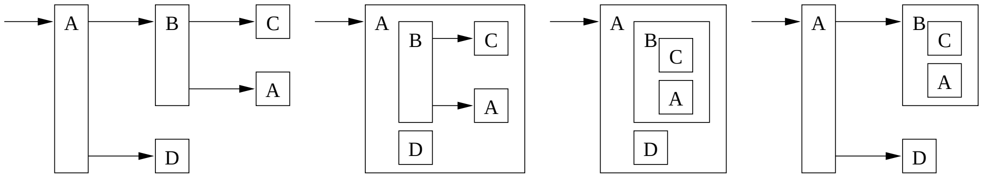

Figure 10: Schematic view of underlaying

tree can be collapsed. In addition, a user can specify elisions for the system to use on a per-class basis. (By default, the system automatically flattens associative classes (Section 4.3.7), and it underlays metaclasses in Smalltalk and standard library classes in C++.) Users can distinguish these classes at a glance, making the diagrams easier to assimilate and interpret.

3.7 Charts as Subtrees

Collapsing and expanding subtrees lets the user navigate the program execution step by step. Clicking on raised rectangles one after the other lets the user explore the execution in increasing detail. But without a clear destination, the user is unlikely to uncover specific behavior through navigation alone.

Searching and filtering are invaluable in that respect, but we also provide visual guidance through several alternative renderings (or **charts**) for collapsed subtrees. For example, a subtree may appear as a class legend (Figure 11) showing the kinds of objects in that subtree. When this is insufficient, the user can choose a more detailed chart showing a class communication graph (Figure 12). Not only can he tell that a particular class of objects participate in a given subtree, but he also can see the classes with which they interact in the immediate context.

Other chart metaphors are possible, of course. A meter showing accumulative CPU time, an inter- or intra-class call matrix [9], and a histogram of instances [9] are shown schematically in Figure 13. These reveal subtree information with varying emphases and levels of detail. The pattern view accommodates charts of any size without significant rearrangement: conventional subtrees need only be displaced downwards.

4 Generalization

So far we have described some of the interactive features of the view that let the user expand, elide, and extract execution information. These are all visual manipulation techniques; the view supports nonvisual techniques as well. The view can be searched in several ways, and it can detect, generalize, and save recurring execution patterns. These capabilities work synergistically with the visual manipulation and navigation techniques, giving the user powerful tools for understanding program execution.

4.1 Why bother detecting recurrences?

We've already seen one kind of recurrence in Section 3.4, namely iteration. Execution patterns of iterative behavior rarely justify the space they consume. Combining them produces smaller, easier to understand views—just compare Figures 5 and 6. In most cases, Figure 5 provides little information of consequence over Figure 6, yet Figure 6 is considerably easier to assimilate. Not only does it reduce clutter; it makes the iteration explicit even as it highlights the recurring computation.

The execution patterns that result from iteration and recursion aren't merely similar; they are

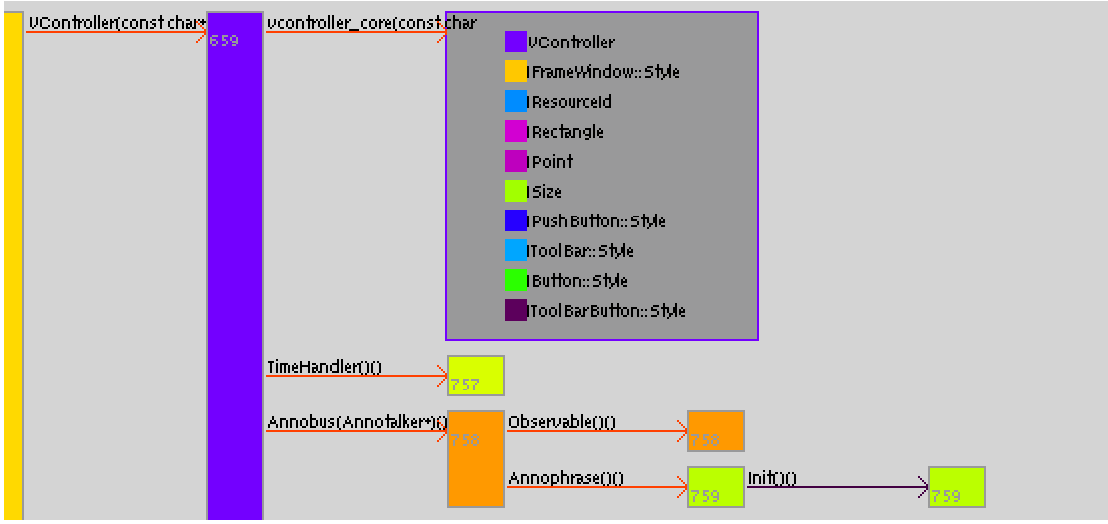

Figure 11: Class legends

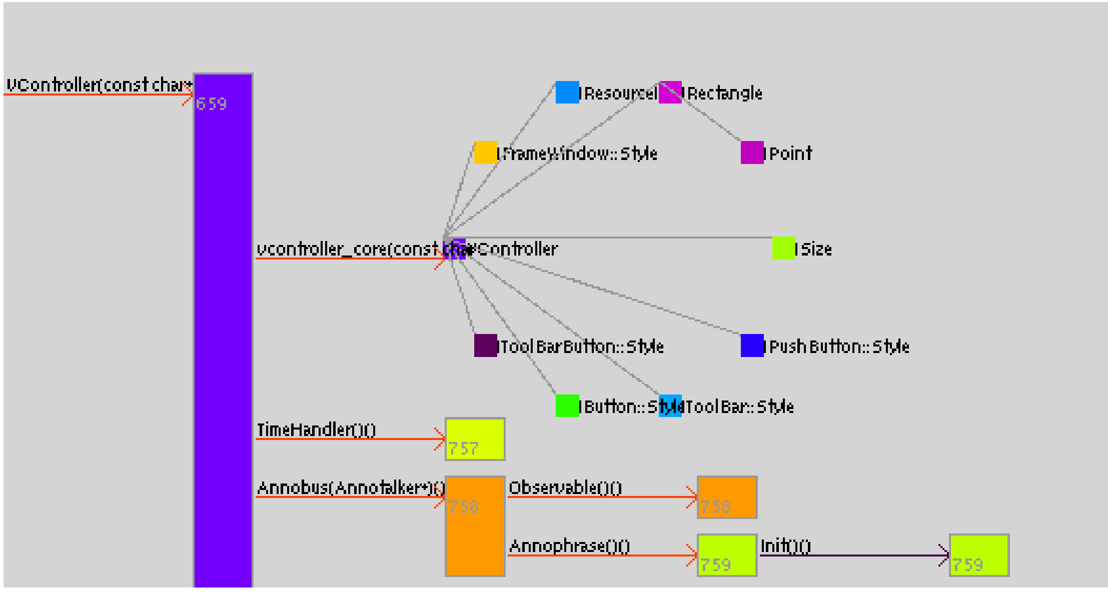

Figure 12: Class communication graph

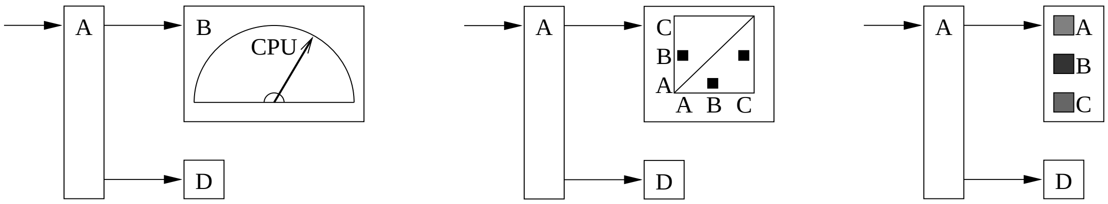

Figure 13: Other chart metaphors

grouped together. Iterative execution patterns stack vertically, while recursive execution patterns chain horizontally. But recurrence isn't limited to iteration and recursion. Widely separated patterns may be similar or identical as well (due to parameterized types, for example). The more widely separated they are, the more likely a user is to miss the recurrence.

Automatic pattern detection is particularly important here. The system can attract our attention to far-flung yet similar patterns. Looking closely at the zoomed-out view of Figure 8, for example, you might notice recurring splotches of color at distant intervals. A bird's-eye view makes it *possible* to see these distant recurrences, but it doesn't make doing so easy or accurate.

When you see an interesting pattern, you'll probably want to find similar occurrences. Conversely, if you've seen a pattern and never want to see it again (because you're already familiar with it, or it does not concern you), then you'd want to hide its recurrences. The techniques we've described for filtered expansion (Section 3.3) filter per class or per method. Filtering *per pattern* requires automatic recognition of pattern recurrences.

4.2 Why generalize?

Merely spotting recurrences is of limited use unless patterns are also **generalized**—that is, unless we can identify *inexact* recurrences among patterns. Searching for recurrences without generalization is like searching text without regular expressions: every match must be exact. We found that our first prototype, which did not generalize, didn't always combine patterns as we thought it should, because it didn't recognize a repetition of similar but not identical patterns. Often the differences among such patterns are immaterial; they are the same for the programmer's intents and purposes.

In fact, relatively few execution patterns in a typical program will be absolutely identical—involving the same objects, messages, and message order. Hence strict matching is too restrictive. Consider again the pattern shown in Figure 5, in which the sequence `more`, `cur`, `update`, and `next` appears to recur six times. These recurrences are not really identical, however, because each repetition involves a different receiver of the `update` message. Without generalization, these patterns would not be identified as recurrent (as in Figure 6).

But generalization promises more than just a better job of recognizing loops. Measuring the average performance of just one execution pattern, for example, is probably meaningless for optimization purposes. Not much time or space will be saved by optimizing one pattern. But the opportunities for improving performance expand considerably the more patterns recur. Generalization can identify potentially numerous recurrences that would otherwise be missed.

4.3 When are patterns the same?

To make generalization effective, we studied the situations in which two or more patterns might be deemed equivalent. That led to algorithms that could be tuned for different pattern matching criteria. As a result, the user can modulate generalization of patterns into higher-order patterns, thereby controlling the amount of information displayed. Seemingly complicated interactions often reduce to a handful of higher-order patterns, saving the user the trouble of examining multiple, trivially differentiated interactions.

The following are the generalization criteria we found to matter most to programmers: identity, class identity, message structure, depth-limiting, repetition, polymorphism, associativity, and commutativity. A user can choose one or more criteria with which to combine patterns automatically.

4.3.1 Identity

The simplest definition of similarity considers two execution patterns identical if their topologies are identical and if there is a perfect match for every object and every message in both patterns. This is easy to check, but it's too strict in general—few patterns match under this definition. Matching on identity is most useful for finding exact matches—that is, when the programmer knows precisely what to look for.

4.3.2 Class Identity

Often, the same pattern of messages affects different sets of objects. If the same message gets sent to the same class of object, the programmer will interpret the sends as identical behavior. Therefore it's useful to consider messages sent to different objects as part of the same pattern if the rest of the pattern is the same *and* the receiving objects are of the same class. We emphasize the similarity of such patterns visually by associating colors with class types. Two patterns match with respect to class identity if they have the same graph structure and identical coloring.

Consider the execution pattern associated with instantiating an object in Smalltalk [11] by executing `Rectangular new`. The pattern involves the class object `Rectangular` (of the metaclass `Class`) and a newly allocated object of class `Rectangular`. Distinct instantiations do not follow *identical* patterns because each involves a particular instance of `Rectangular`. But to the programmer, these instantiations are almost always identical conceptually. Thus it's useful to characterize all of them with a more general pattern, which we might call the "Rectangular instantiation" pattern.

More precisely, if two patterns have isomorphic invocation graphs (with messages labeling the edges and types labeling the nodes), then they can be considered instances of the same pattern. Most programmers will consider such patterns similar and will prefer to see the generalized pattern. Nevertheless, when we present a generalized pattern, we still allow the user to query the actual object identity by browsing through different instances of the pattern in the trace.

This technique is applied, for example, in depicting repetitions. If you look at Figure 6 closely you will notice that the green and black objects no longer show an object identifier. That's because they represent multiple instances, as shown in Figure 5.

4.3.3 Message Structure

In a weakly typed language like Smalltalk, the type of the message receiver is not always known a priori. While the Rectangular instantiation pattern matches instantiation of different `Rectangular` objects, instantiation of a `Rectangular` and a `Circle` would not be considered the same, even though both invoke the same method `new` defined in the metaclass `Class`. We can generalize the matching by considering two patterns the same if their message structure is identical and there is a nontrivial[1] color substitution from one's coloring to the other. This resembles the notion of alternations in regular expressions.

[1] An example of a "trivial" color substitution would be one that maps all colors to black, which has the same effect as ignoring color altogether.

4.3.4 Depth-Limiting

Encapsulation is a key concept in object-oriented programming. It lets a programmer think in terms of what an object *does* (i.e., its interface), not *how* it does it (its implementation). Encapsulation works at multiple levels, especially in large, layered systems. Objects in one layer communicate with objects in the layer below them exclusively through their interfaces.

Such layering provides another criterion for controlling matches. If a program is designed with good layering and encapsulation, patterns involving lower-level objects should be independent of those involving higher-level objects. Thus we can apply different matching criteria to patterns on the basis of message depth. A simple approach ignores matches to patterns beyond a given depth in the tree.

Continuing our Smalltalk example, the Rectangular instantiation pattern begins with a message `new` received by the class object `Rectangular`. This invokes the method `new` defined in `Class`. In response, `Rectangular` sends itself `new:` to allocate a new object, and then it forwards the message `new` to the new object. That in turn invokes `Rectangular>>new`. Object creation may therefore span a deep execution graph during its initialization. By limiting the depth of the instantiation pattern, we can match instantiations comprising varied initialization patterns.

4.3.5 Repetition

Suppose one invocation of a loop executes it 1000 times, and a different invocation executes it only 999 times. (This behavior might be characteristic of a search algorithm that looks for a word somewhere in a long string, for example.) Most programmers would consider the overall behavior of the two invocations to be pretty similar. The loop would appear as a repeated structure along the vertical dimension (characteristic of iteration) in the execution pattern view. An example of repetition in the horizontal direction (recursion) could be an instantiation of object lists, where each object instantiates its neighbor. A list of six objects should reflect the same instantiation pattern as would a list of eight objects.

The matching algorithm can ignore the number of repetitions in a lower-level pattern when matching at a higher level. Hence if two patterns contain invocations of the search algorithm on different words, they may be considered instances of a more general pattern. This resembles the notion of repetition in regular expressions.

4.3.6 Polymorphism

Consider a program that draws a series of shapes, such as rectangles and circles. This can show up in an execution pattern as an iteration of polymorphic `draw` messages to different `Shape` objects. Rather than showing the classes of these objects, we match all these subclasses according to their common base class `Shape`.

Returning once more to the Smalltalk example, `Circle new` and `Rectangular new` generate polymorphic instantiation patterns provided both are subclasses of `Shape`. Of course, matching two classes when their base class is the same does not imply that we will match two different method names. This suggests a more general definition of polymorphic matching that will match methods with overloaded names as well.

4.3.7 Associativity

What if the colors of two patterns match but their structures don't? Structural equivalence can take into account properties such as associativity of objects or methods.

If a mathematical function φ is *associative*, then nested invocations can be flattened by effectively removing the inner sets of braces: $\varphi(x, \varphi(y, z)) = \varphi(\varphi(x, y), z) = \varphi(x, y, z)$. Mathematica [23], for example, uses this property to get functions into a standard form before matching them to patterns: `Flatten[`$\{\{C, \{D\}\}, E\}$`]` flattens out the sublists at all levels, resulting in $\{C, D, E\}$. In matching execution patterns, we can treat an "associative" class A in a manner analogous to the braces of the nested list above. Consider the leftmost pattern in Figure 14. Removing the inner calls of A would allow it to match the rightmost pattern in that figure, wherein the first call to A calls C, D, and E directly.

When execution patterns are matched with respect to associativity of classes, method labels are ignored. Conversely, we can choose to ignore object type and/or identity and match patterns on associativity of methods exclusively. If a method is associative, then a sequence of recursive calls matches a single call. Here, a call to some other (re)definition of the same (inherited) method is considered a "recursive" call.

Associativity can help suppress successive self-invocations and recursive method invocations. For example, one may want to ignore all private method calls in class A because A's implementation of a public service is of no concern.

Recall (Section 3.6) that the execution pattern view let the user choose how a class is displayed based on its attributes. A natural rendering (and the system's default) for expressing associativity is flattening, as shown in Figure 14. Note how C, D, and E are aligned vertically as a result of flattening A. The depth of the tree no longer represents *physical* stack depth but rather a *logical* one—one that ignores self-invocations. This example also demonstrates how we carefully match the elision technique—flattening in this case—to the need, namely association.

Figure 14 also reveals a difficulty. Had C and D (or D and E, or both) been the same class, flattening A would make them match based on repetition.

4.3.8 Commutativity

Commutativity means that a pattern wherein A is called first and then B matches a pattern wherein B is called first and then A. To avoid comparing every combination of calls whenever a commutative match is desired, objects acquire their commutative characteristics from their context: two subtrees match on commutativity if they involve instances of the same set of classes. This limits the possible matchings to complete subtrees. Thus if our visualization displays subtrees as orderless lists of classes as depicted in Figure 11, then two subtrees that match based on commutativity will be rendered identically.

5 Implementing Generalization

A tracing tool collects "method enter" and "method leave" events from the target program as it runs. The resulting trace may drive visualizations in either real-time or post-mortem.

From the information in the trace, the view builds a tree structure representing the sequence of messages and the objects that receive them. Every node in the tree structure may represent an execution pattern. To recognize, classify, and generalize execution patterns, the view assigns a hash value to every node in the tree structure. A set of tree nodes reflecting a generalized pattern will have identical hash values.

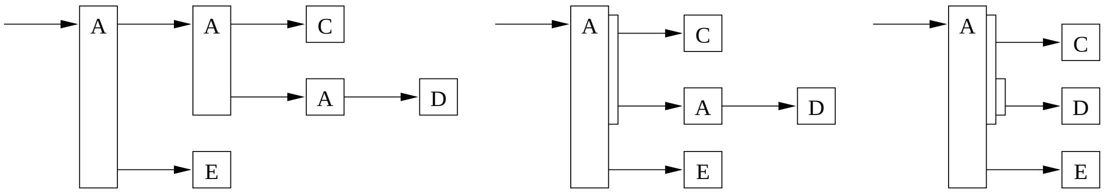

Figure 14: Using flattening to visualize association

A recursive hash function computes a hash value for each subtree in a single pass over the structure. The hash value of a given subtree is a function of its children's hash values and one or more values in the subtree's root. Which values are used depends on the matching criterion. If we want to match on method names, the values include the method and class names. If we want to match on class names only, then the method name is omitted.

Similarly, if we want to match on a particular object, then that object's ID must be included. If we're interested in knowing not the total number of method calls but just the methods that got called, we can ignore nodes representing redundant calls during recursion. If we don't care about the order of calls, we can sort the nodes first.

Using a Universal_2 class of hash functions [6] ensures that computing the hash value recursively will produce a good distribution. The chance of two different patterns producing the same hash value is no more than the depth of the tree divided by the maximum hash value. Most hash values used in this computation are too large to use as an index into a table—typically around 2^{32}. So each hash value is masked down to a reasonable number of bits for an index.

The masked hash value is registered in a **pattern dictionary**, which contains all the hash values. Each entry in this dictionary corresponds to a (perhaps trivially) generalized execution pattern. When the system encounters a hash value that's already registered, it increments one or more fields associated with the dictionary entry for that value—for example, a **frequency** field recording the total number of incidences of this pattern. Other cumulative information such as CPU time may be stored to help further characterize and differentiate the program's execution patterns.

For good performance, our prototype checks only that hash values are identical to establish equivalence. This can create false positives, but the distribution is good enough to make this rare, and performance is considerably improved over more detailed comparisons.

Sometimes it is impractical to show method names in graphs due to a lack of space, an unworkably large number of methods, or both. We use two tricks to mitigate this problem: zooming and "flyover." It's possible to zoom the view without scaling the fonts, providing more room for method names at higher magnifications. We also identify the method (and/or object) directly under the mouse cursor in separate status panels. When space is at a premium, we omit method names and other labeling and let the user examine methods selectively by "flying over" them with the mouse.

6 Related Work

Our work is a confluence of two research areas: pattern matching and program visualization. Both have long histories, but they have been independent until recently.

Tree and string matching is one of the most thoroughly documented areas of research in computer science [13, 21, 5, 19, 7, 4, 1]. The algorithms are many, they are well-understood, and they have been applied widely—in interpreters for nonprocedural languages; optimizing compilers; algebraic computation; sorting, searching, and differencing facilities of all types; and automatic theorem proving. We have tailored a few of these algorithms to the needs of our domain. Pattern extraction has in fact been applied to programs before, but almost exclusively as a static analysis (as in SCRUPLE [20], for example).

We can classify most object-oriented visualization systems into two categories: *macroscopic* and *microscopic*. Macroscopic systems collect and present cumulative execution information, while microscopic systems help you understand the sequence of message sends between objects. Our earlier visualization work is characteristic of the macroscopic type [9, 10]; the microscopic variety is more common [17, 12, 8, 2].

This dichotomy reflects how hard it is to present execution information effectively. Macroscopic systems condense execution information to make it more manageable, much like traditional profiling tools do—and they discard much information in the process. Our original visualization system offered several views of message sends per class, the overall activity of objects, and resource (CPU and memory) consumption. These cumulative views could be navigated to uncover more detail—for example, total message traffic at the method (rather than class) level. Still, this didn't provide much insight into how the program accomplishes a particular task, such as initialization or screen update. We were discarding execution information at the object level in the name of scalability.

Microscopic systems show the sequence of messages between objects, potentially yielding deep insights about small sections of the program. But microscopic systems have the opposite problem from their macroscopic counterparts: too much detail limits scalability. Showing individual objects sending messages to each other (typically using a nodes-and-arcs motif) quickly gets unworkable as the number of objects increases. Not only is there a multitude of indistinct objects, but their communication is equally obscure. The messages and messaging paths are lost in a jumble of lines and bubbles. Any benefits of visualization are quickly lost.

Only recently have people tried to narrow the gulf between these extremes. Program Explorer [18] is a representative example; it implements Jacobson's interaction diagrams nearly verbatim to visualize object interaction. The system employs several filtering techniques to help manage large numbers of objects. But since every object in the visualization occupies a column from the top to the bottom of the view, scaling remains a problem. Moreover, the interaction diagram becomes unwieldy when objects communicate with others that were created much later, again because rails appear in the order of object creation. Hence when there is communication between objects created at distant intervals, the screen fills with long horizontal lines spanning potentially many screenfuls. This problem is not easily solved. Rearranging the vertical lines that denote objects is highly disruptive; grouping the objects reintroduces the shortcomings of macroscopic systems.

Our execution pattern view offers many of the advantages of both micro- and macroscopic approaches. It can provide cumulative or global execution information by collecting and generalizing patterns throughout the execution trace. It also lets the user inspect the program at any level of detail with its navigation (elision and expansion) features. It is not perfect—accurate pattern generalization being the most challenging aspect—but it does offer some new choices on the micro-to-macroscopic visualization spectrum.

7 Conclusion

Execution patterns enhance object-oriented visualization technology in three ways. First, they offer an intuitive and scalable metaphor for object communication. Our execution pattern view's structure is based on a natural notation, one that captures object interaction clearly and lends itself to interactive manipulation. One can easily navigate the execution trace to survey the interactions of objects, classes, and methods. The pattern view differs from previous views in that it scales smoothly to make even lengthy interactions of objects intelligible. Elision and expansion mechanisms make the journey from the macroscopic to the microscopic easy for users.

The second enhancement comes from generalizing similar execution patterns. Generalization lets us abstract away redundant behavior automatically, even when the redundancy is imprecise or non-periodic.

Third, execution patterns give us a foothold for characterizing system complexity. While the definitive complexity metric for execution patterns is far from obvious, we have found even simple-minded metrics (e.g,. pattern redundancy) useful for pinpointing not just any program hotspots but those that are most likely to have a simple remedy.

We have implemented the execution pattern view and integrated it into *Ovation* [9, 10], our research prototype for object-oriented program visualization. The views in this paper were taken from traces of Ovation itself. The system can visualize any C++ or Java program using traces generated from the VisualAge development environment [14]. Ovation also supports visualization of Smalltalk programs. To generate Smalltalk traces, we added instrumentation to the Little Smalltalk [3] and VisualAge Smalltalk [15] environments.

We have experimented with traces from mid-size programs such as Ovation itself and truly large systems such as Taligent. We found that the view and mechanisms described here were helpful in uncovering unexpected behavior, in understanding unfamiliar code, and in improving performance.

Currently we are making the pattern matching facility more flexible. We are augmenting the matching criteria with a pattern equivalent of regular expressions to provide a semiformal way to express similarity. We are also exploring the potential of visual grammars, especially their synergy with the current visualizations. Finally, we plan to report qualitative results of execution pattern visualization as we learn through user feedback and controlled experiments.

Acknowledgments

We are grateful to our colleague Doug Kimelman for providing us a Smalltalk tracing facility by instrumenting the VisualAge Smalltalk environment.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] H. Böcker and J. Herczeg. Browsing through program execution. In *INTERACT '90*, pages 991–996. Elsevier Sci. B.V., 1990.

[3] T. Budd. *A Little Smalltalk*. Addison-Wesley, Reading, Massachusetts, 1987.

[4] J. Burghardt. A tree pattern matching algorithm with reasonable space requirements. In *Proc. 13th Colloquium on Trees in Algebra and Programming*, volume 299 of *LNCS*, pages 1–15, Mar 1988.

[5] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up tree pattern matching. In *Proceedings of CAAP*, pages 72–86, 1990.

[6] L. J. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18:143–154, 1979.

[7] D. R. Chase. An improvement to bottom-up tree pattern matching. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 168–177, Munich, Germany, Jan. 1987.

[8] W. Cunningham and K. Beck. A diagram for object-oriented programs. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 361–367, Portland, Oregon, USA, Sept. 29–Oct. 2 1986. OOPSLA '86, ACM SIGPLAN Notices 21(11) Nov. 1986.

[9] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 326–337, Portland, Oregon, USA, Oct. 23-27 1994. OOPSLA '94, ACM SIGPLAN Notices 29(10) Oct. 1994.

[10] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In M. Tokoro and R. Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming*, number 821 in Lecture Notes in Computer Science, pages 163–182, Bologna, Italy, July 4-8 1994. ECOOP '94, Springer Verlag.

[11] A. J. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[12] V. Haarslev and R. Möller. A framework for visualizing object-oriented systems. In N. Meyrowitz, editor, *Proceedings of the 5th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 237–244. OOPSLA/ECOOP '90, ACM SIGPLAN Notices 25(10) Oct. 1990, Oct. 21-25 1990.

[13] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, Jan. 1982.

[14] IBM. *VisualAge C++, http://www.software.hosting.ibm.com/ad/cset/csetos2/wp1-fam.html*.

[15] IBM. *VisualAge for Smalltalk, http://www.software.hosting.ibm.com/software/ad/vastub.html*.

[16] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, Massachusetts, 1992.

[17] M. F. Kleyn and P. C. Gingrich. GraphTrace—understanding object-oriented systems using concurrently animated views. In N. Meyrowitz, editor, *Proceedings of the 3rd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 191–205, San Diego, California, Sept. 25-30 1988. OOPSLA '88, ACM SIGPLAN Notices 23(11) Oct. 1988.

[18] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 342–357, Austin, Texas, USA, Oct. 1995. OOPSLA '95, ACM SIGPLAN Notices 30(10) Oct. 1995.

[19] F. Luccio and L. Pagli. Approximate matching for two families of trees. *Information and Computation*, 123:111–120, 1995.

[20] S. Paul. SCRUPLE: A reengineer's tool for source code search. In *Proceedings of the 1992 IBM CAS Conference*, pages 329–345, Toronto, Ontario, Nov. 1992.

[21] R. Ramesh. Nonlinear pattern matching in trees. *Journal of the ACM*, 39(2):295–316, Apr. 1992.

[22] R. E. Sweet. The Mesa programming environment. *ACM SIGPLAN Notices*, 20(7):216–229, July 1985.

[23] S. Wolfram. *The Mathematica Version 3*. Wolfram Media and Cambridge University Press, third edition, 1996.