

Modeling Object-Oriented Program Execution

Wim De Pauw, Doug Kimelman, and John Vlissides

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

Abstract

This paper describes a way of organizing information about an object-oriented program's execution. The context is our system for visualizing that execution. The critical constraints are *completeness*, *compactness*, and *efficient retrieval*. We describe our design and how it meets these constraints.

1 Introduction

Much is known about how to characterize programs statically. Contemporary programming languages embody countless lessons learned over nearly a half century of modern computing. It's clear that a language must balance expressiveness and simplicity, abstraction and specificity, flexibility and robustness. The proportions can vary according to need, but programming language design is necessarily a compromise between conflicting goals.

We can think of a programming language as defining a vast space of legal programs relatively few of which are of any practical use. To define a space we must have orthogonal dimensions. In the case of structured programming languages we might devote one dimension to data structures, another to control flow, and another to procedural decomposition. A contour in the resulting space defines a unique combination of data structures, control flow constructs, and procedures that characterize a specific program. Other programming models might define the space in different terms. For example, classes, inheritance, and polymorphism provide orthogonal bases for object-oriented languages.

In general, programmers use a language to map their ideas into a contour representing a viable program in the space, one that solves their problem. From this perspective, programming tools like compilers, browsers, and editors exist to realize, visualize, change, or otherwise manipulate the mapping into the program space. Tools for manipulating the program complete a feedback loop that lets a programmer view and modify the mapping to suit his needs. Figure 1 illustrates how one maps his conceptual understanding into a program in the space defined by a programming language.

The feedback loop's effectiveness determines how quickly and easily one can compose a useful program. The path along the loop has four key components:

1. A person's conceptual understanding of a problem.
2. A space in which to express the problem and its solution.
3. A model of the problem in the expression space.
4. Tools for viewing and manipulating the model in the expression space. We call this activity **navigating** the expression space.

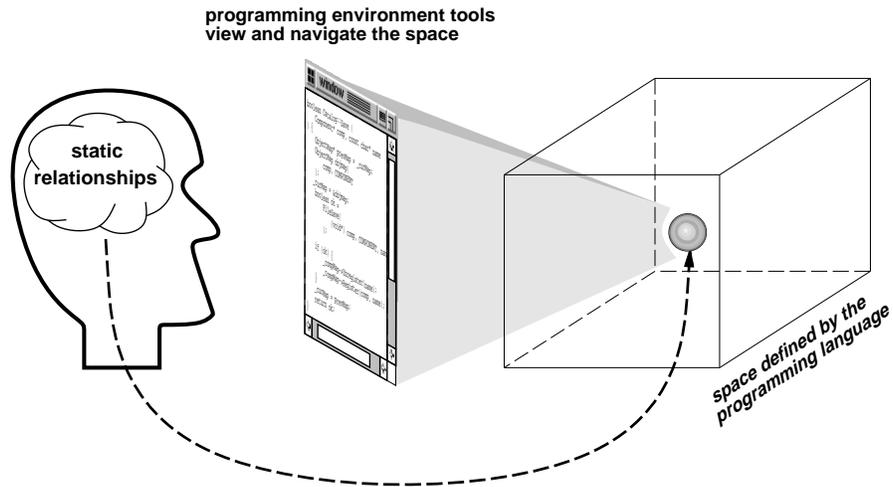


Figure 1: Mapping conceptual relationships into the program space

In this perspective the development process begins and ends with one’s understanding of a problem, but that doesn’t mean one must understand the problem *completely* before proceeding further. Such understanding is neither mandatory nor attainable in practice, simply because proving the completeness of a set of requirements is infeasible for most real-world problems. Without a guarantee of completeness, one cannot be assured that an analysis is viable prior to its use in the design process.

That argues against separating design and analysis and treating them as independent activities. In reality they are closely related and should be done concurrently and iteratively. Hence program development—that is, navigating the expression space—is inherently an iterative process. The mark of good tools is that they are useful for design, implementation, *and* analysis. We eliminate artificial distinctions between these activities when we think of tools in terms of navigating the expression space.

This perspective is also useful because it has a direct analog in characterizing the *dynamic* aspects of a program. A program’s dynamic behavior is just as important to its design, implementation, and refinement as its static specification. This is especially true of object-oriented programs, where the gulf between static specification and run-time behavior is particularly wide. But while much is known about the static aspects of programs, much less is known about characterizing and manipulating their dynamic aspects. An understanding of dynamic aspects is critical for building tools that let one visualize, change, or otherwise manipulate a program’s dynamic behavior.

Figure 2 shows a feedback loop analogous to that of Figure 1 but characterizing a program’s dynamic rather than static aspects. The path along the loop has the same four components we introduced for the static case, but their interpretation is different. Here, conceptual understanding concerns the program’s run-time behavior as opposed to its static structure. People often think in these terms, so this analogy should be obvious.

The other three components, however, have no analogs to current languages and tools. For example, the orthogonal criteria that make up the dynamic expression space include time, the call stack, and, in the case of object-oriented programs, message sends and method binding. The model in the expression space comprises a succession of events that characterize the program’s execution. There are no established techniques for capturing such information. Consequently, few tools exist for navigating this space compared to tools for navigating the static space.

We have sought to address these three components of the loop by

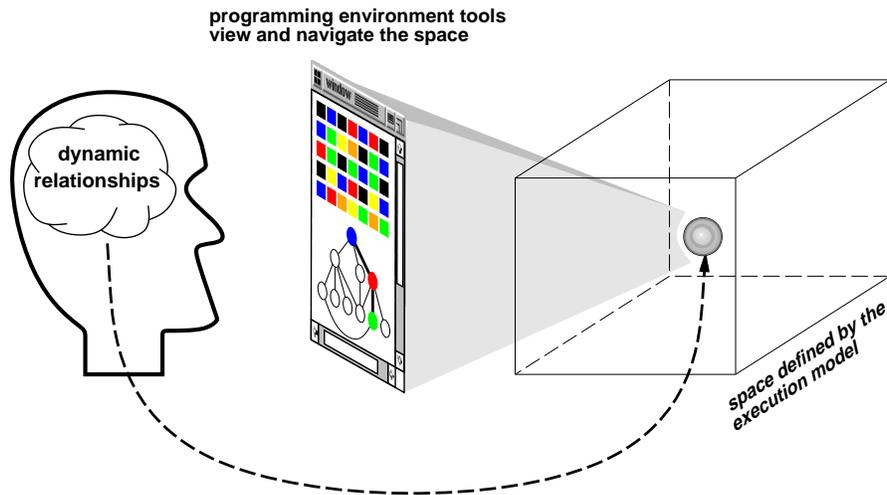


Figure 2: Mapping conceptual relationships into the execution space

- defining an **event** space for characterizing the dynamic aspects of programs,
- modeling the event space in concrete terms, and
- building tools for navigating the event space.

In this paper we describe our approach to characterizing the dynamic behavior of object-oriented programs. The primary goals of our approach are threefold:

1. *Completeness*. Capture as much information as possible on significant aspects of a program's execution, minimizing information loss.
2. *Compactness*. Use as little storage as possible to hold the captured information.
3. *Efficient retrieval*. Arrange execution information for easy and quick access so that navigation tools can operate with minimal overhead.

This work is the cornerstone of a system we have built for visualizing the execution object-oriented programs [11]. We support C++ programs currently, but the design could be easily retargeted to other object-oriented languages. The system comprises an architecture for building visualization applications and a prototype implementation. We have developed numerous dynamic views that present different aspects of a program's execution in object-oriented terms. The infrastructure described in this paper plays an important part in making these views possible and effective.

We start by defining the event space more specifically. We then describe how we model this space, including the tradeoffs we have made to achieve the goals of completeness, compactness, and efficient retrieval. Next we demonstrate the model's utility by showing how several visualization tools use the model to generate their displays. We conclude with a review of related work, a summary of this work, open issues, and our plans for the future.

2 The Event Space

We can characterize a program's execution as a succession of interesting events. The frequency and granularity of events depends on the level of detail we want to capture. For example, characterizing

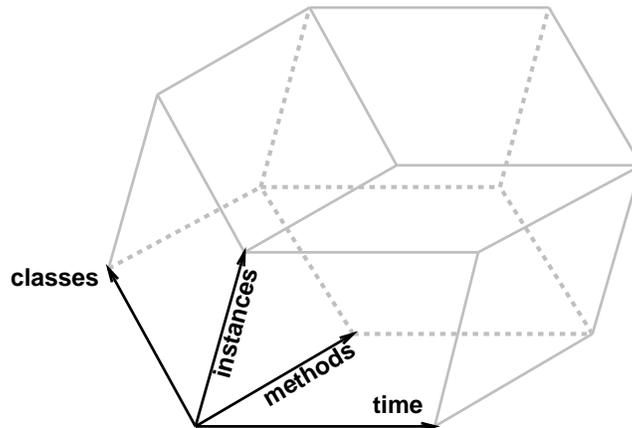


Figure 3: Canonical four-dimensional event space

a program's execution at the instruction level would produce a flood of events that will be difficult to store, manage, and interpret. On the other hand, collecting only a few coarse-grained events gives us little insight into the program's execution.

Choosing an appropriate level of abstraction is therefore critical to characterizing program execution accurately. Object-oriented programmers think in terms of classes, instances, inheritance, messages, methods, and so forth. Interesting events in this context might include object construction and destruction, message sends, method invocation and return, among others. By capturing program execution in terms of these artifacts, we obtain a close match between our run-time characterization and the programmer's mental model of the program.

We organize these artifacts into a four-dimensional space (Figure 3) having axes for classes, instances, methods, and time. Each increment along the class axis corresponds to a class in the program. The classes are arranged in order of instantiation. Likewise, increments along the instance axis refer to instances in the program. The method axis reserves two increments per method in the program, one for entrances into the method and another for exits from the method. The time axis defines a program-independent variable that measures execution progress.

Thus a point in the space is described by the coordinate quadruple

$$(class, instance, method, time)$$

Each point corresponds to an event during program execution. The program generates a stream of these events as it runs, thereby forming a contour in the space. We populate the space by noting four important events:

1. A **construction** event marks a class' instantiation via a construction method.
2. A **destruction** event marks the destruction of an instance via a destruction or finalization method (if any).
3. An **enter** event records method entry resulting from a message send to an instance of a class.
4. A **leave** event records method exit.

The canonical four-dimensional space is a convenient metaphor for representing and analyzing an object-oriented program's execution. We can extract information by traversing or projecting one or more dimensions of the space in different combinations to produce subspaces. We can

reconstruct the call stack and compute message frequency, instance and method lifetimes, and conventional profiling statistics. Moreover, by coupling the information in the space with static program information, we can analyze class and instance relationships to determine the degree to which classes rely on code inheritance, for example, and how activity is distributed within aggregate objects. Such computations are fundamental to tools for navigating the event space.

3 Modeling the Event Space

The canonical space is a useful conceptual model, but a straightforward implementation would be impractical for two reasons. First, a typical program can generate many hundreds of thousands of construction, destruction, enter, and leave events. Storing every such event in the space would be prohibitively expensive. Second, the space is organized for generality and as such is suboptimal for retrieving two common kinds of information: **differential** and **integral**. In this section we describe how we model the canonical space to address these problems.

3.1 Differential and Integral Information

It is common for visualization tools to update their appearance on every event. Events arrive at a rapid rate, perhaps many thousands per second, so event processing overhead must be kept to a minimum. Obviously it isn't possible to scan the entire event space on each event to determine its impact on the tool's display presentation. Thus it must be possible to ascertain *differential* information, that is, the specific incremental change since the last event.

Consider a tool that displays a graphical representation of the total number of invocations on every instance. If the tool stores this information implicitly in the graphics it displays, then only differential information (e.g., "another message *foo* sent to instance *x*") is required to update the appropriate graphic(s) on each message send. Providing differential information of this sort is simple enough: simply deliver events to tools as they arrive.

In contrast to differential information, *integral* information is cumulative in nature. Examples of integral information include the number of times an object received a particular message (e.g., "total sends to instance *x*: 23"), how often an object sent a message to another object, and how many times a class has instantiated objects of another class. We would need information like this to construct a different graphical representation halfway through a run. Acquiring these statistics over time generally requires a traversal over part or all of the canonical event space. A more efficient organization is required to make these queries more efficient.

3.2 Call Frames

Rather than storing each event explicitly as in the canonical event space, we store *combinations* of events in objects called **call frames**. Different combinations offer varying trade-offs between modeling accuracy and efficiency. Our goals were to store execution information compactly and allow fast retrieval of integral information.

Developers of object-oriented programs often need to trace patterns of communication between objects. For example, suppose we want to discover who allocated a given instance. We can find out by searching the event space for the object that produced the construction event for that instance. At any time τ (except in functions such as C++'s `main` function or static member functions), a method f of class t executes for a receiver instance i of class c . We express this with the notation

$$c :: i . t :: f \tag{1}$$

denoting an instance-method combination. c and t usually refer to the same class unless f is an inherited method. For example, if we have an object `mylist` of the class `List`, we can express an invocation of the `next()` method on this object as follows:

$$\text{List} :: \text{mylist} . \text{List} :: \text{next}()$$

If we consider the stack of method calls where each stack frame¹ records an instance and a function, then combination (1) will represent the top of the stack at time τ .

If we want to know at some point which method is executing on an object, then we need the information recorded in a combination like (1). For example, to determine which construction method created `mylist`, we should look for all matching combinations (1) for which $i = \text{mylist}$ and f is a construction method.

Now suppose combination (1) was produced by a message from another instance I of class C as a result of executing method F from class T at $\tau - 1$:

$$C :: I . T :: F \tag{2}$$

If “ \rightarrow ” means “calls,” then we can write the full message send as

$$C :: I . T :: F \rightarrow c :: i . t :: f \tag{3}$$

This is an example of a call frame. At time τ , combination (2) is just below combination (1) on the stack.

Call frames contain the information we need to trace and compile statistics on message sends. Returning to our example, suppose the object `mylist` is owned by an instance `mytable` of class `HashTable`. Invoking a `printAll()` method on `mytable` might cause `mytable` to invoke the `next()` method on `mylist`:

$$\begin{array}{l} \text{HashTable} :: \text{mytable} \quad . \quad \text{HashTable} :: \text{printAll}() \rightarrow \\ \text{List} :: \text{mylist} \quad . \quad \text{List} :: \text{next}() \end{array} \tag{4}$$

If we want to find out how often the `HashTable` class’ `printAll()` method calls `List`’s `next()` method, we count the occurrences of call frame (3) for which $T = \text{HashTable}$, $F = \text{printAll}()$, $t = \text{List}$, and $f = \text{next}()$. Note how a call frame holds information spanning two consecutive points in time: the invocation of the caller $C :: I . T :: F$ and the invocation of the callee $c :: i . t :: f$. In contrast, combinations like (1) represent a single invocation $c :: i . t :: f$.

3.3 Storing Call Frames

We recover call frames from the event stream and store them as the program runs. However, storing every call frame separately would still be expensive both in space and in time, especially when computing queries for integral information. Instead we store only unique call frames and the accumulated number of occurrences of each. This approach greatly reduces storage and integral query costs at the expense of differential information.

For every possible call frame (3) we store a count of occurrences in an eight-dimensional matrix. We express a look-up into this matrix with the notation

$$E(C, I, T, F, c, i, t, f) \tag{5}$$

For example, matrix entry

$$E(\text{HashTable}, \text{mytable}, \text{HashTable}, \text{printAll}(), \text{List}, \text{mylist}, \text{List}, \text{next}())$$

¹ A conventional *stack* frame is not the same as a *call* frame. A stack frame records a function call and its actual parameters, whereas a call frame stores a superset of this information, as we explain below.

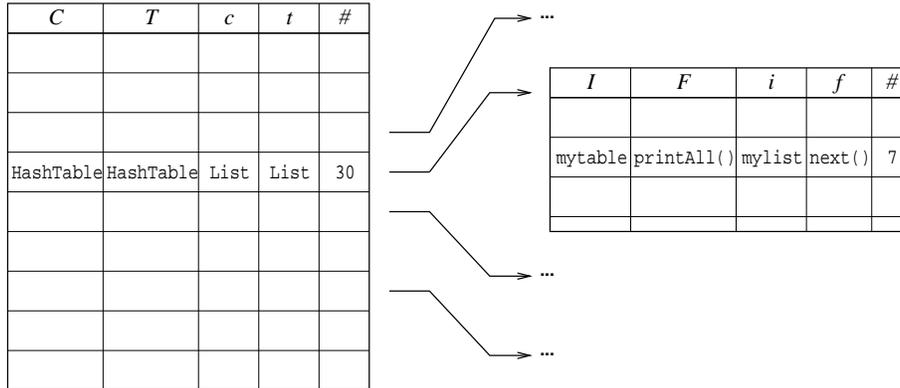


Figure 4: Internal organization of eight-dimensional call frame matrix

would store the number of occurrences of call frame (4) above. Whenever a new event arrives, we update E by incrementing the value for the corresponding call frame (5) by one.

E is normally very sparse, because relatively few combinations of class names, instances, and methods are possible. Moreover, few of all possible caller-callee pairs make sense given the program's static structure. To exploit these properties, we implement the matrix as a two-level hash table (Figure 4). The first level of this data structure is a hash table organizing the information in terms of classes. Given values C , T , c , and t , the entry $E_{c,t}(C, T, c, t)$ stores the accumulated number of call frames having the form of (3) above but with I , F , i , and f left unspecified. The number reflects the total number of messages sent from all instances of one class to all instances of another. In our example,

$$E_{c,t}(\text{HashTable}, \text{HashTable}, \text{List}, \text{List})$$

stores the number of times `HashTable` objects sent a message from within any of their own (i.e., uninherited) methods to any `List` object (thereby invoking any uninherited `List` method).

For each nonzero $E_{c,t}(C, T, c, t)$ there is a second-level hash table $E_{i,f}$. Given values I , F , i , and f , then entry $E_{i,f}(I, F, i, f)$ stores the accumulated number of call frames (3) above. The sum of all entries in $E_{i,f}$ equals the count stored in $E_{c,t}(C, T, c, t)$.

$E_{i,f}$ provides finer-grain information on call frames for specific instances of C and c and for specific methods of T and t . In our hash table example the second level hash table for

$$E_{c,t}(\text{HashTable}, \text{HashTable}, \text{List}, \text{List})$$

contains more specific information about the instances and methods involved in the communication between the `HashTable` and `List` classes. For example, entry

$$E_{i,t}(\text{mytable}, \text{printAll}(), \text{mylist}, \text{next}())$$

stores the total number of messages sent *from* the `mytable` instance within `HashTable`'s `printAll()` method *to* `mylist`, thereby invoking the `next()` method from the `List` class.

3.4 Queries

The two-level structure is well-suited to tools that let the user navigate from a macroscopic perspective of the program's execution to a more microscopic one. Specifically, instances almost always outnumber classes by a wide margin. Therefore it is usually appropriate to navigate the event space at the class level initially to avoid swamping users with information. From this level a user is in

a good position to request more detailed information at the method or instance levels. Because a user already has certain classes in mind when making method or instance queries, the two-level hashing ensures these are as fast as queries at the class-level.

While the eight-dimensional matrix model allows highly specific queries, most tools require less specific information, especially when the user works at a macroscopic level. Thus queries on the eight-dimensional matrix usually include one or more wildcards. For example, to determine how many times the `List` class' `next()` method has been called by `HashTable`'s `printAll()`, we would search E for all the call frames that match

```

C = I = c = i = *
T = HashTable
F = printAll()
t = List
f = next()

```

The sum of the values from all matching frames gives the number of calls from `HashTable`'s `printAll()` to `List`'s `next()`.

Another common query involves searching call frames for specific fields as opposed to simply adding up the values from matching call frames. Suppose we want to determine which object created another object `myobject` from class `List`. We must search E for the call frame satisfying

```

C = I = T = F = *
c = t = List
i = myobject
f = List()

```

where `f = List()` corresponds to the construction method. The call frame whose C, I, T, F fields match the query's will have c, i, t, f fields that reveal the creator of this object.

Now suppose that `List` is a subclass of class `Persistent`. In this case we might want to know if a particular `List` object ever uses a method inherited from `Persistent`. This information can help us assess quantitatively how much code inheritance is exploited in a program. A query specifying

```

C = I = T = F = i = f = *
c = List
t = Persistent

```

will give us the call frames that show which inherited methods are invoked on which instances and by whom.

It is also possible to trace the complete set of outgoing messages for a particular class. A query specifying

```

I = T = F = c = i = t = f = *
C = List

```

reports the call frames of all messages produced by the `List` class. Conversely, we can find the set of incoming messages with a query specifying

```

C = I = T = F = i = t = f = *
c = List

```

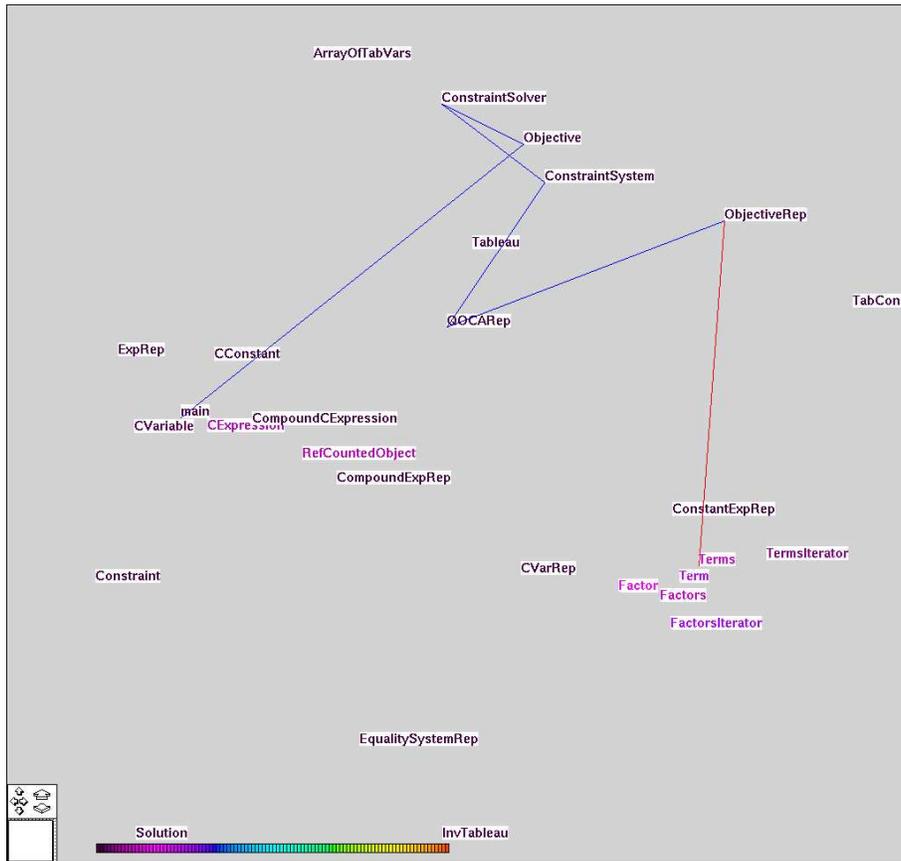


Figure 5: Inter-class call cluster

4 Visualizing the Event Space

Our model of the event space provides a basis for a variety of navigation tools. We have developed a set of dynamic views that let a user visualize different parts of the event space. The views are based on a software architecture that supports rapid development of visualization applications. We describe the architecture's design, implementation, and application elsewhere [11]. Here we show how several views use the event space model for display and navigation.

4.1 Visualizing Communication

The **inter-class call cluster** provides a dynamic overview of communication patterns between classes. Figure 5 shows a snapshot of this view during a program's execution. This view shows class names as floating labels. The amount of communication between instances of two classes determines the distance between their labels. The view is animated so that the more communication there is between classes, the more their labels gravitate towards each other and cluster together. Classes that communicate infrequently repel each other.

This view also indicates the current call stack by showing the classes of methods on the call stack. A blue path leads from the label `::main` through each of these classes. The last segment of the path, leading to the currently active class, is red. In Figure 5 the thread of control goes from

::main, through Objective, ConstraintSolver, ConstraintSystem, QOCARep, and ObjectiveRep, and finally to the currently active class Terms.

The inter-class call cluster focuses attention on the most active and most cooperative classes at any moment. These classes provide a good starting point for more detailed study either for optimization or understanding the structure of an application—clustered classes, for example, are likely to be tightly coupled or from the same subsystem [14]. The number of classes in a cluster is typically small, on the order of ten classes or fewer, probably because systems with broader interactions are exponentially more complex and are less likely to be developed in the first place.

This view uses an iterative, force-based node placement algorithm. Two kinds of forces work on each floating node. First, every pair of nodes experiences a repulsive force that is inversely proportional to the distance between them. Second, every pair is also affected by an attractive force proportional to the total number of messages sent between the corresponding classes.

When this view is displayed initially, it must determine the number of messages exchanged so far between every pair of classes. Since information about instances or member functions is not required for this view, it suffices to query $E_{c,t}$ for all the possible call frames at the class level only:

$$C = T = c = t = *$$

A query with these parameters will return the total number of messages between any pair (T, t) of classes. Once the view has accumulated query results for every combination of classes, then it can start the node placement algorithm to position its class name labels. From then on, the view extracts differential information directly from the event stream. The view uses this information to update its appearance incrementally as the program executes.

4.2 A Closer Look at Communication

While the inter-class call cluster offers insight into the dynamic messaging behavior of the program, the **inter-class call matrix** (Figure 6) gives cumulative and more quantitative information. Classes appear on the axes in the order in which they are instantiated. Base classes always appear closer to the origin than their subclasses. A colored square in this visualization represents the number of calls from a class on the vertical axis to a class on the horizontal axis. The color key along the bottom indicates relative number of calls. Colors range from violet, denoting fewer calls, to red, denoting more calls.

The inter-class call matrix offers a different perspective of the information shown in the inter-class call cluster. Consequently, the inter-class call matrix makes the same kinds of integral queries and differential updates. An additional feature of this view is its “zooming” capability, which lets a user see more detailed information on demand. When a user clicks on the square for a class A on the left and a class B at the bottom, another view appears, the **inter-function call matrix**. This subview displays the frequency of calls from individual methods of A to methods of B . Figure 7 shows the subview produced by clicking on a square in Figure 6.

To get the details of calls from methods of A to methods of B , the inter-function call matrix queries E with the following values:

$$\begin{aligned} C &= I = F = c = i = f = * \\ T &= A \\ t &= B \end{aligned}$$

The model reports the cumulative number of calls between any pair of methods F and f from classes A and B , respectively. Once the inter-function matrix reflects the current values, it makes differential updates whenever A sends a message to B .

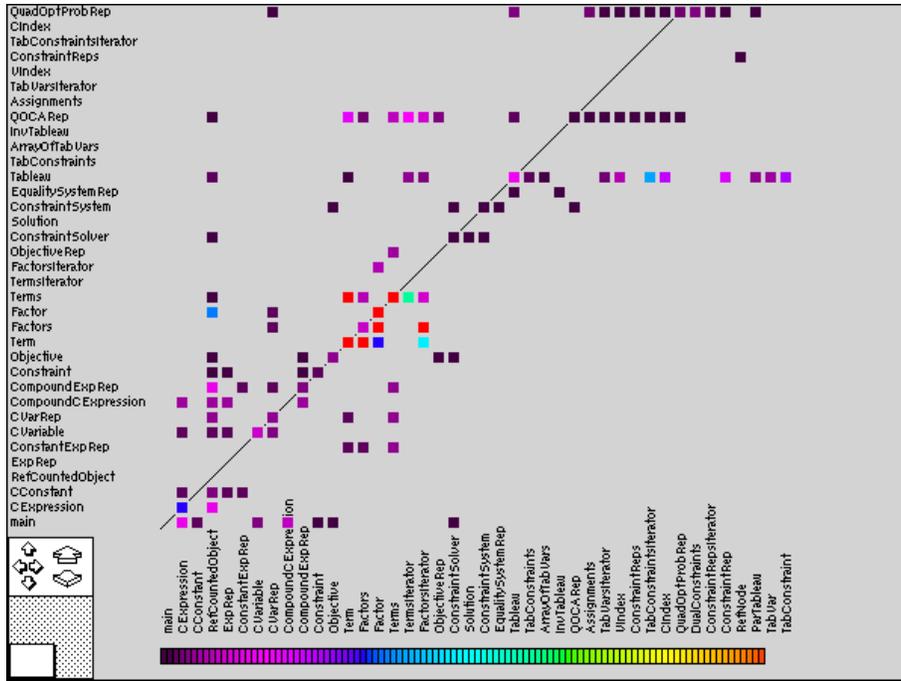


Figure 6: Inter-class call matrix

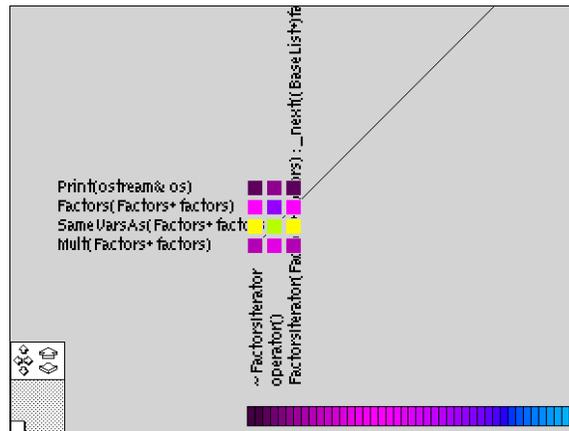


Figure 7: Inter-function call matrix subview

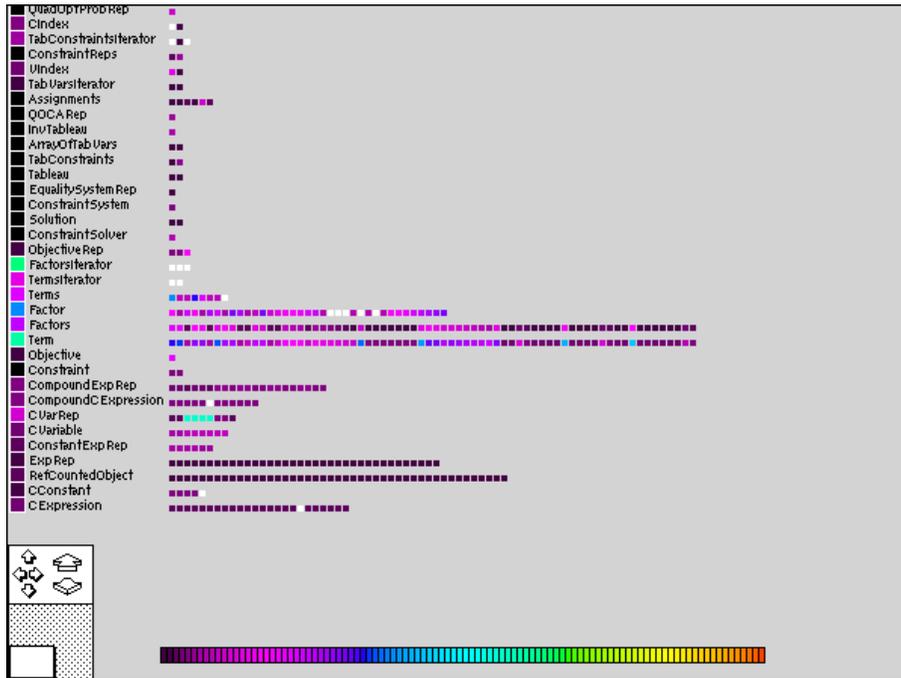


Figure 8: Histogram of instances

4.3 Insight from Instances

The preceding visualizations emphasize the display of relationships between *classes*. Focusing on *instances* can reveal program dynamics at finer levels of granularity.

The **histogram of instances** (Figure 8) displays all instances of each class. Rows of small colored squares form the bars of the histogram. Each bar represents all instances of the class whose label appears to its left. Again, a square's color indicates the number of messages an instance has received. Colored squares appear and disappear as objects are instantiated and destroyed. White squares indicate objects that have been destroyed; these squares will be reused by newly created instances. This visualization lets us see how many instances exist at a given time and their level of messaging activity. It also shows relative object lifetimes and anomalies such as undesired copy constructor calls in C++ that are manifest as extremely short-lived objects.

We determine the current set of instances of a class A using a query with the values

$$\begin{aligned}
 C &= I = T = F = i = * \\
 c &= t = A \\
 f &= A()
 \end{aligned}$$

where $A()$ is the construction method. Differential information keeps the view updated thereafter as instances are created and destroyed.

Like the inter-class call matrix, the histogram of instances can also furnish more detail about the instances it depicts. For example, clicking on a particular instance a of class A produces a subview that displays three sets of information:

1. The messages that this instance received, from whom, and how often.
2. The messages that this instance sent, to whom, and how often.

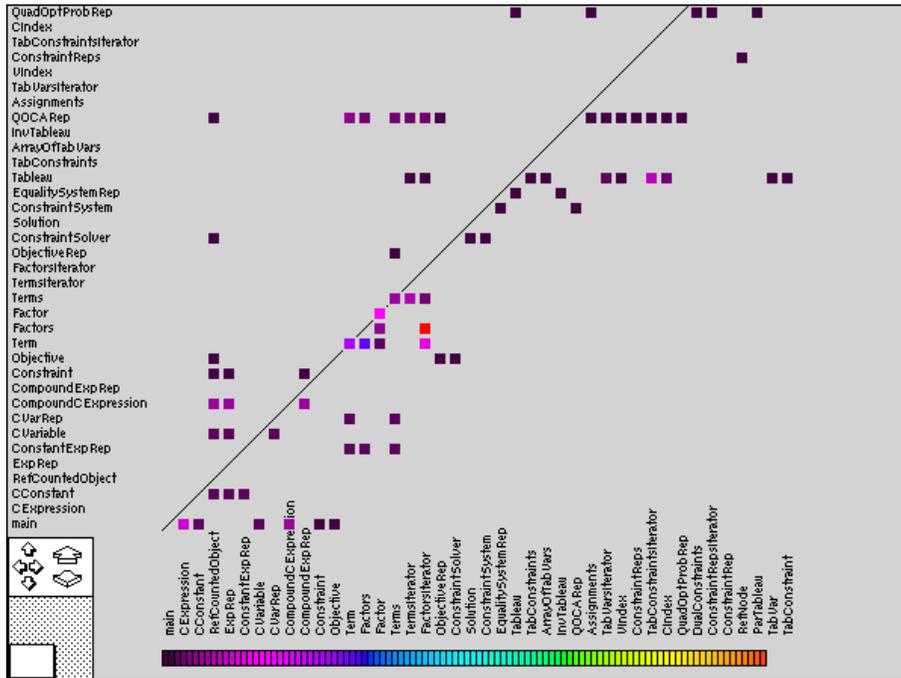


Figure 10: Allocation matrix

plots the classes that allocate new objects versus the classes they instantiate. This view shows allocation dependencies and the most frequently allocated objects at the class level. We can use this information to pinpoint the sources of allocations and subsequently reduce storage and construction costs in the application being visualized. The allocation matrix collects allocation data via the query parameters

$$C = I = T = F = c = i = t = *$$

$$f = X()$$

for all construction methods $X()$ in the program.

The allocation matrix provides a zooming capability as well. For example, clicking on the (Term, Factors) square in Figure 10 will produce the subview shown in Figure 11. The subview displays the allocation patterns of Factors instances by Term instances, with details at the method level. The view shows us how many instances of Factors were created by methods of Term, and which construction methods were used. In general, the necessary query parameters are

$$C = I = F = i = *$$

$$T = A$$

$$t = B$$

$$f = B()$$

where A is the allocating class, B is the class that is instantiated, and $B()$ is B 's construction method.

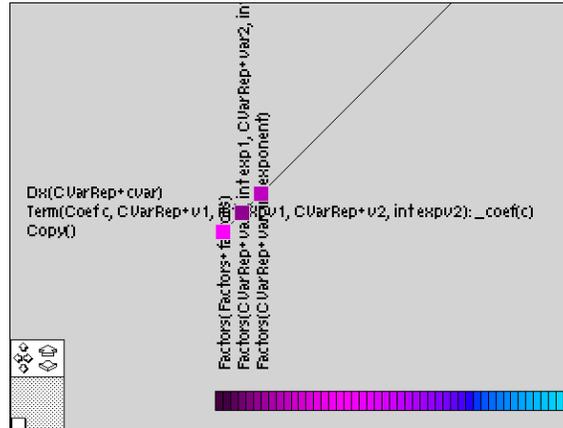


Figure 11: Allocation matrix subview

5 Related work

Many have recognized the need for monitoring and visualizing program execution as an adjunct to traditional language-oriented tools [7, 2, 10, 8]. Most work has centered on visualizing non-object-oriented languages, as we discuss elsewhere [11]. Relatively little has been reported on execution modeling, especially for object-oriented languages.

Perhaps closest to this work is that of Snodgrass [13], who presents a method for monitoring program execution in which a user expresses all aspects of navigation—the events of interest, the modeling criteria, and presentation—with expressions in a relational algebra. The set of events is stored in a “conceptual database” that can be “queried” for execution information. The key difference between this approach and ours is that queries in Snodgrass’ system must be specified before execution starts, and no information is ever actually stored. The choice of event data to collect and process is entirely query-dependent. Specifying the queries up-front makes it possible to compute the queries as the program executes; hence the database is “conceptual” and not real. This approach necessitates repeated executions until the user can narrow to the relevant points in the program.

Domingue [5] proposes visual representations of both behavioral and performance aspects for rule-based programs. His monitoring system measures time (for performance monitoring) and counts rules (for generating behavioral visualizations) to provide a complete picture of the program. An interesting aspect of this work is its ability to navigate from coarse-grain views to finer-grain views showing individual rule firings.

Bruegge, Gottschalk, and Luo [3] describe an object-oriented framework for dynamic analysis of distributed programs. The focus of this work is on how to collect and distribute the streams of events coming from different nodes. The events are then interpreted in real time through one or more views. However, the programs being monitored are not themselves object-oriented, and there is no repository for event information that can be queried to construct new views for navigation.

Systems for visualizing the execution of object-oriented programs have concentrated more on presenting execution information than on organizing and storing it. The GraphTrace system by Kleyn and Gingrich [9] is a tool for visualizing the execution of programs written in Strobe, an object-oriented language based on CommonLisp [12]. Visualization is done in two passes. First, the system constructs structural views called “graphs” that represent object invocation graphs or method invocation graphs. The graphs are animated in the second pass by highlighting the invocation path, thereby depicting execution behavior. GraphTrace does not include mechanisms for navigating the large quantities of data generated during execution, and query capabilities

are limited to inspecting the attributes of individual objects. These drawbacks are common to contemporary object-oriented visualization systems [6, 4, 1].

6 Conclusion

A program's dynamic aspects are just as important to its development as its static specification. Like traditional programming environment tools that navigate the static specification space, tools for navigating the execution space need a semantic basis. Object-oriented concepts like classes, instances, messages, and methods offer a sound basis not just for static modeling but for dynamic modeling as well. In a sense these concepts are *more* important to dynamic modeling, because they offer a way of managing the vast amount of information that can characterize even a simple program's execution.

The model we have described here reflects but one set of design choices for characterizing the execution of object-oriented programs. Our overriding goal has been to capture execution information as accurately as possible with reasonable storage and execution costs. We have had to make inevitable compromises to achieve this goal, but the result is general and flexible enough to support a variety of visualizations. We have described four in this paper; others are described elsewhere [11], and more are under development.

Several aspects of this work merit further attention. The model as it is implemented currently stores integral information only as far as the instance level. No information is accumulated on an object's instance variables, for example. A third hashing level could cache this data, thereby providing quick access to additional information of potential interest to programmers. Moreover, we only *store* information currently; there is no support for *changing* run-time values and thereby affecting execution. Such a capability would be useful in testing "what-if" hypotheses during debugging, for example. We also plan to explore query optimization techniques to further reduce modeling overhead. Through these efforts we hope to improve our model of object-oriented program execution and consequently the visualization tools that depend on it.

Acknowledgments

We thank Richard Helm for his seminal contributions both in the model and in implementing the various visualizations.

References

- [1] H.D. Böcker and J. Herczeg. Browsing through program execution. In *INTERACT '90*, pages 991–996. Elsevier Science Publishers B.V. (North Holland), 1990.
- [2] M.H. Brown and R. Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, 1985.
- [3] B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analyzers. In *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 65–82, 1993.
- [4] Ward Cunningham and Kent Beck. A diagram for object-oriented programs. In *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 361–367, 1986.
- [5] J. Domingue. Compressing and comparing metric execution spaces. In *INTERACT '90*, pages 997–1002. Elsevier Science Publishers B.V. (North Holland), 1990.

- [6] V. Haarslev and R. Möller. A framework for visualizing object-oriented systems. In *ACM OOPSLA/ECOOP '90 Conference Proceedings*, pages 237–244, 1990.
- [7] C.F. Herot, G.P. Brown, R.T. Carling, M. Friedell, D. Kramlich, and R.M. Baecker. An integrated environment for program visualization. In H.-J. Schneider and A. J. Wasserman, editors, *Automated Tools for Information Systems Design*, pages 237–259. North Holland Publishing Company, 1982.
- [8] D.N. Kimelman and T.A. Ngo. The RP3 program visualization environment. *The IBM Journal of Research and Development*, 35(6), November 1991.
- [9] M.F. Kleyn and P.C. Gingrich. Graphtrace—understanding object-oriented systems using concurrently animated views. In *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 191–205, 1988.
- [10] B.A. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *ACM CHI '86 Conference Proceedings*, pages 59–66, Boston, MA, April 1986.
- [11] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the behavior of object-oriented systems. In *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 326–337, 1993.
- [12] Reid Smith, Paul Barth, and Robert Young. A substrate for object-oriented interface design. In Bruce Shriever and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.
- [13] R. Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.
- [14] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.