Exploiting UML dynamic object modeling for the visualization of C++ programs

Brian A. Malloy*
Computer Science Department
Clemson University

James F. Power†
Computer Science Department
National University of Ireland

Abstract

In this paper we present an approach to modeling and visualizing the dynamic interactions among objects in a C++ application. We exploit UML diagrams to expressively visualize both the static and dynamic properties of the application. We make use of a class diagram and call graph of the application to select the parts of the application to be modeled, thereby reducing the number of objects and methods under consideration with a concomitant reduction in the cognitive burden on the user of our system. We use aspects to insert probes into the application to enable profiling of the interactions of objects and methods and we visualize these interactions by providing sequence and communication diagrams for the parts of the program under consideration. We complement our static selectors with dynamic selectors that enable the user to further filter objects and methods from the sequence and communication diagrams, further enhancing the cognitive economy of our system. A key feature of our approach is the provision for dynamic interaction with both the profiler and the application. Interaction with the profiler enables filtering of methods and objects. Interaction with the application enables the user to supply input to the application to provide direction and enhance comprehension or debugging.

Keywords: Unified Modeling Language, sequence diagram, aspect oriented programming, program comprehension

1 Introduction and Motivation

The current trend in the construction of large scale, multi-version systems is to exploit object technology to profit from its advantages in reuse, extensibility, and ease of maintenance over the procedural approach to application development. However, the object oriented paradigm comprises an interesting blend of powerful constructs and unique challenges in comprehension, debugging and testing of the application. In the procedural realm, run-time program behavior can be traced using a debugger, where code is executed on a line-by-line basis. However, much of the complexity in object oriented software lies in the interactions among the objects and methods, rather than in the statements within the methods [Jacobs and Musial 2003; Reiss 2003]. Although object and method interaction can be tracked using a traditional statement-level debugger, the scale and complexity of these interactions provides compelling motivation for higher level visualization techniques.

An increasingly accepted approach to modeling object oriented applications is the Unified Modeling Language, or UML, which provides a selection of graphs and diagrams for capturing static and dynamic behaviors of an object oriented application [Rumbaugh et al. 1999]. One of the primary driving forces behind the evolution of version 2.0 of the UML standard has been "an approach to developing software that shifts the focus of development from code to models, and to automatically maintaining the relationship between the two" [Selic 2004]. Many modern code development environments already espouse this goal to some degree, offering integration between class diagrams and code projects, or use-cases and testing strategies. However, there is relatively little research on the exploitation of UML diagrams for visualizing the dynamic behavior of objects and methods and even less research on linking UML diagrams describing dynamic aspects of software, such as sequence diagrams and communication diagrams, with the run-time behavior of the code they describe.

In this paper we present an approach to modeling and visualizing the dynamic interactions among objects in a C++ application. We exploit UML graphs and diagrams, typically used during the requirements and design phases of the life cycle, to expressively visualize both the static and dynamic properties of the application. One of the central issues in dealing with run-time properties of objects and methods, and one that makes the domain particularly suitable to visualization, is the quantity of data involved. Even relatively small programs can easily generate thousands of objects and millions of method calls in a relatively short running time. In our approach, we make static use of a class diagram and call graph of the application to select the parts of the application to be modeled, thereby reducing the number of objects and methods under consideration, with a concomitant reduction in the cognitive burden on the user of our system. We use aspects to insert probes into the application to enable profiling of the interactions of objects and methods and we visualize these interactions by providing sequence and communication diagrams for the parts of the program under consideration. We complement our static selectors with dynamic selectors that enable the user to further filter objects and methods from the sequence and communication diagrams, further enhancing the cognitive economy of our system.

An important contribution of our work is the construction of a toolset, SPIDOR[1], which facilitates our investigation into the pragmatics of our approach. A unique feature of our work is that rather than using trace data generated from program executions [Jacobs and Musial 2003; Jerding et al.

*e-mail: malloy@cs.clemson.edu
†e-mail: power@cs.may.ie

[1] SPIDOR is a mnemonic for *Selection and Profiling of the Interaction of Dynamic Object Representations.*

1997; Orso et al. 2003; Reiss 2003], we allow the user of SPIDOR to dynamically interact with both the profiler and the application during live execution. Interaction with the profiler enables filtering of methods and objects. Interaction with the application enables the user to supply input to the application to provide direction and enhance comprehension or debugging. Since games and other graphics applications pose unique challenges to comprehension and debugging, our case study usage of SPIDOR is an arcade game implemented with the Simple Directmedia Layer, SDL [Pazera 2003].

In the remainder of this paper we describe the design of our approach and the implementation of SPIDOR, highlighting the practicalities involved in event selection and run-time profiling. Section 2 briefly reviews the terminology used in this paper. Section 3 presents an architectural overview of our work, including some of the issues relating to tracing C++ programs. Sections 4 and 5 present our Selector and Profiler tools respectively. Section 6 surveys the existing work in the field of visualizing object oriented applications, particularly in relation to UML diagrams and object-level visualization. Section 7 concludes the paper and describes future directions of our work.

2 Background

Graph representations of program structure, control flow and data flow, have long been a part of software development. For example, a *call graph* consists of nodes representing call sites or functions and edges representing a function invocation [Murphy et al. 1998]. Call graphs have been used for comprehension and optimization of both procedural and object oriented applications.

Recently, the object oriented approach has become widely accepted due to its promotion of reuse, extensibility and easier maintenance of large systems. To facilitate modeling of object oriented applications, the Unified Modeling Language, UML, has become widely adopted and utilized to express different aspects of an object oriented system from different viewpoints and at different stages of the development life cycle [Rumbaugh et al. 1999]. The UML consists of thirteen diagrams of which the most commonly used diagram is the *class diagram* [Cook and Brodsky 1999], a graph whose nodes are classes and whose edges express relationships or dependencies between the classes. Class diagrams capture information about the static structure of an object oriented system. The UML also provides interaction diagrams that capture behavior or interactions among the class instances, or objects, in an object oriented system. The interaction diagrams that are important in our work include sequence and communication diagrams.

A *sequence diagram* is an interaction diagram that models time along the vertical length of the diagram and method invocations between objects along the horizontal length of the diagram. Sequence diagrams are the most commonly used interaction diagram and are the object oriented counterpart of a call graph. The *communication diagram* is a second interaction diagram that shows instances of classes, their relationships, and the flow of messages between the instances [Ambler 2004]. Communication diagrams are similar to sequence diagrams with a difference that communication diagrams allow free placement of participants where connecting links representing messages can be adorned with nested decimal numbers to resolve ambiguity with self-calls. Further discussion of this numbering can be found in Section 4.3, with an illustration in Figure 7.

An *object diagram* is a form of interaction diagram that is similar to a class diagram with a difference that the nodes are instances rather than classes [Jacobs and Musial 2003]. Thus, an object diagram is sometimes called an instance diagram. An object diagram is usually a special case of a class diagram or a communication diagram [Ambler 2004].

In our approach to visualization, we use the call graph and class diagram as static program representations that enable the user to make fine-grained and coarse-grained selections of the parts of the program to be profiled. We then provide an interactive animation of the program during execution using sequence and communication diagrams to illustrate the program behavior. We provide further filtering of the sequence and communication diagrams as part of our dynamic filtering to reduce the cognitive effort of the user of our system.

3 System overview: Tracing C++ Programs

In this section we describe the principle technical details relating to the SPIDOR toolset, including the Selector and Profiler tools. Although the focus of this paper is the visualization of C++ code, almost all of the development was carried out in Python [Rossum 2003]. Python proved ideal for the kind of text-processing required for the static aspects of the tools, as well as the GUI development using the Tkinter module, which provides an interface to the Tk GUI toolkit. Python's rapid prototyping facility and ease of interoperability with other tools were also important factors.

While a significant amount of software development takes place in C++, the tools and techniques for analyzing, profiling and visualizing this software are relatively less developed than those for other object oriented languages. For example, Java programs are easy to parse, either at source code or bytecode level, and can be profiled either through JVM modification, inserting probes in the bytecode or, since version 1.4 of the SDK, using a built-in profiler API. In contrast, C++ is notoriously difficult to parse [Knapen et al. 1999; Malloy et al. 2003], and does not have a rich, robust, standardized run-time environment comparable to the JVM.

Figure 1 provides an overview of the system architecture of the SPIDOR toolset, with the static features illustrated on the left and the dynamic features illustrated on the right side of the figure. The input to the system is illustrated in the lower left corner of the figure, an application written in C++. The output of the system is illustrated on the right side of the figure where we provide visualization of sequence and communication diagrams on a Tk GUI canvas. We use the *gcc* abstract syntax tree (AST) as our internal representation of the C++ application; we provide more detail of this phase of SPIDOR in Section 3.1.

The rectangle in the center of Figure 1 represents the Selector component of the SPIDOR system. The Selector is written in Python and takes the *gcc* AST as input via our *pyGast* API, constructing a class diagram and system call graph for the application.

The user can then select the classes, methods or method invocations to be profiled, and, as explained in Section 3.3, the relevant profiling code is then woven through the application source code using AspectC++ [Mahrenholz et al. 2002]. This is compiled to an executable, in our case a Unix shared object, that interacts with the Profiler to generate and visualize the sequence and communication diagrams for the selected parts of the application.

In the next section we provide more detail about the *gcc*

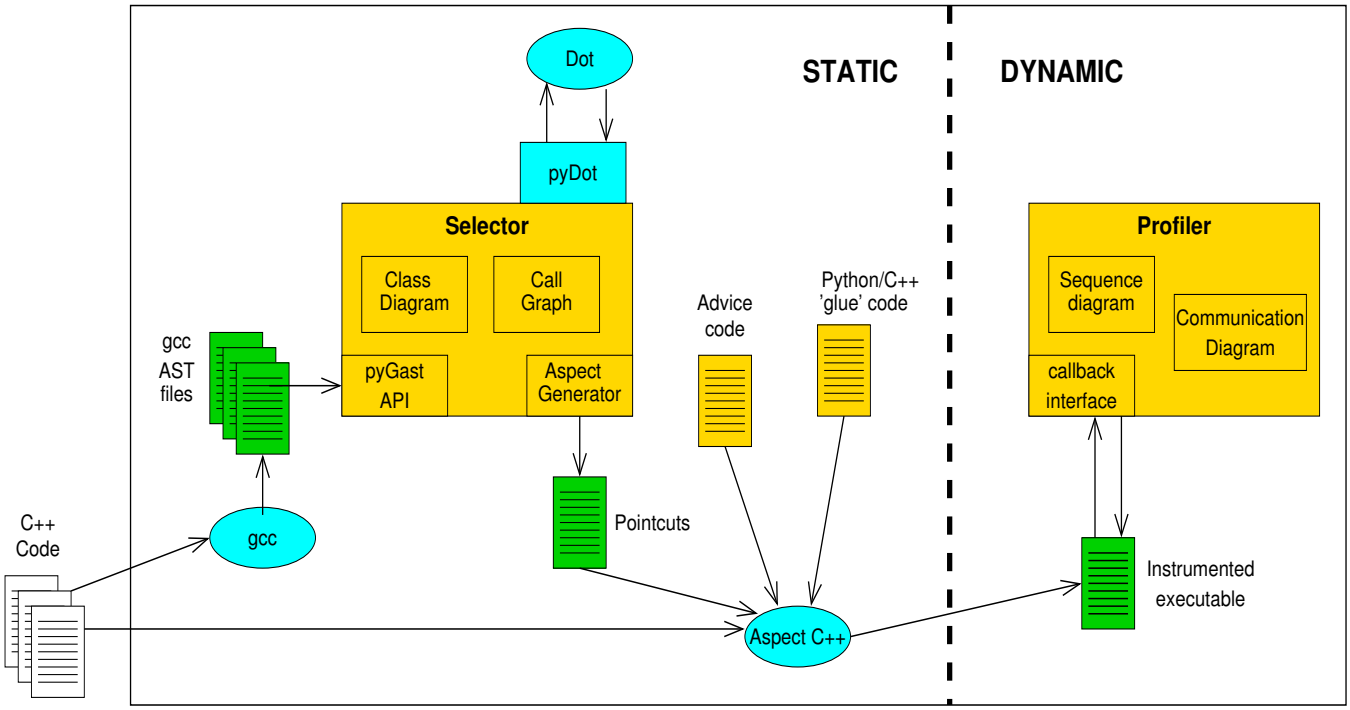

Figure 1: *System architecture.* This figure presents the main components of the SPIDOR toolset. External programs and APIs, such as *gcc* are colored blue, generated code, such as the pointcuts, is colored green, and our own code is colored gold. The C++ code for the program being profiled, shown here in white, is the input to the system.

AST and in Section 3.2 we provide more detail about our use of Dot [Gansner and North 2000]. In Section 3.3 we describe our use of aspects and in Section 3.4 we describe our approach to connecting C++ and Python.

3.1 Interfacing with gcc

The SPIDOR toolset uses the C++ compiler from the GNU Compiler Collection, *gcc*, as its front end. We had previously used *gcc* as a basis for C++ program comprehension by instrumenting its parser [Power and Malloy 2002], but this proved difficult to maintain over various *gcc* releases. Since version 3.0, *gcc* has begun to develop an internal abstract syntax tree (AST) format, known as GENERIC, which provides a high-level representation of a C++ program [Merrill 2003]. This representation is also reasonably accessible, since it can be generated as a text file using a compiler switch (`-fdump-translation-unit-all`).

However, it should be noted that most of the GENERIC documentation is in the form of comments in the *gcc* source code, and some effort is required to disentangle the constructs used. We have written a Python API, *pyGast*, to facilitate working with the *gcc* GENERIC output. *pyGast* provides methods to parse the *gcc* output to produce a Python representation of the AST, as well as providing facilities for visitor-based AST navigation. Our *pyGast* API also builds a representation of the class hierarchy, the call graph, and provides output in text, XML and Dot formats.

3.2 Interfacing with Dot

Visualizing C++ programs statically and dynamically using class diagrams, call graphs and communication diagrams requires the construction of non-trivial graphs. To visualize these graphs we use the Dot tool, part of the *Graphviz* graph visualization software suite [Gansner and North 2000]. We found the default Dot layout strategy, based on a hierarchical layout, to be the most suitable for each of the graphs created. Since the main edges in all three diagrams were based in some way on method calls, the use of the hierarchical layout helped to clearly present the flow of control through the system.

We utilize an existing program, *pyDot*, to represent Dot graphs, and both our Selector and Profiler tools develop this to display Dot graphs in a Python canvas. In particular, our Selector tool provides interpolation between the co-ordinate systems of Python canvas objects and Dot graphs to allow interaction with the Dot graph for the purposes of selecting nodes and edges.

Visualization of sequence diagrams is unencumbered by the layout problem inherent in visualizing call graphs and class diagrams, since sequence diagram sequencing of messages is ordered by time. Thus, we use Dot to visualize our call graph, class and communication diagrams, and we map Tkinter widgets directly onto a canvas to visualize our sequence diagrams.

3.3 Interfacing with AspectC++

In order to create run-time profile data we need to track object creation and destruction, as well as method calls and returns at run-time. To achieve this, we insert probes in the C++ code at appropriate positions, as selected by the user. In order to facilitate using SPIDOR with multiple C++ programs it was important to automate the process of instrumenting the code.

This kind of cross-cutting concern is one of the standard

examples used to promote Aspect Oriented Programming (AOP) [Gibbs and Malloy 2003; Kiczales et al. 1997]. The AOP approach allows for such concerns to be specified separately in Aspects, and these are then woven into the code using a special tool. For this project, we used the AspectC++ compiler [Mahrenholz et al. 2002], an implementation of AOP for C++ that is similar in style to the more widely-used AspectJ.

The parts of the program at which the new behavior is to be added are referred to as *pointcut*, and the behavior itself is called *advice* [Lohmann et al. 2004]. In this project the pointcuts are specific to each profiling instance, and are created using the Selector tool, described in Section 4. The advice in each case involves interfacing with our Profiler tool, and is the same for each program. Thus, the output of the Selector tool is effectively a piece of AspectC++ code specifying pointcuts, which can then be compiled, along with the original code, to generate a version of the program that will interact with the Profiler.

3.4 Interfacing between C++ and Python

Since our Profiler is written in Python, we require a facility to interface with C++. Interfacing with the C++ code being profiled is achieved using the Python/C API, which allows Python code to call C and C++ code, and vice versa. Hence, the advice woven into the code by the AspectC++ compiler also contains hooks to this interface, allowing the Python code to register handlers with the C++ code, which are then called at each pointcut.

We emphasize that this process is automated, and requires no user interaction or user modification of the C++ code. Further, any Python code capable of handling a small set of events can be used in place of the SPIDOR Profiler.

3.5 Case Study: An Arcade Game

In subsequent sections we discuss the operation of our visualization tools, and show examples of their use. The program being visualized on each case is a simple arcade game, written in C++ using the Simple DirectMedia Layer multimedia library. Our approach, and the SPIDOR toolset, is designed work with any C++ program, and we use the arcade game simply as an example.

We chose to profile an arcade game since game software in C++ typically exhibits two properties that make visualization desirable. First, traditional text-based debugging can be difficult to synchronize with graphical or GUI-based software. Second, many games lack design artifacts, such as class diagrams, and an approach that re-generates these automatically from the code can be useful in itself.

The arcade game is written in just under 1,000 lines of C++ code, spread over 9 source files, describing 6 classes in total, with a call graph containing 48 methods and 87 method calls. As an example of the scale of the dynamic visualization problem, running the game for just under 20 seconds leads to the creation of 36,000 objects and just under 273,000 method calls (not including constructors or destructors).

4 Static Visualization: The Selector

In this section we describe the SPIDOR's Selector tool, whose purpose is to visualize the classes and methods in a C++ program, and to allow the user to navigate through these

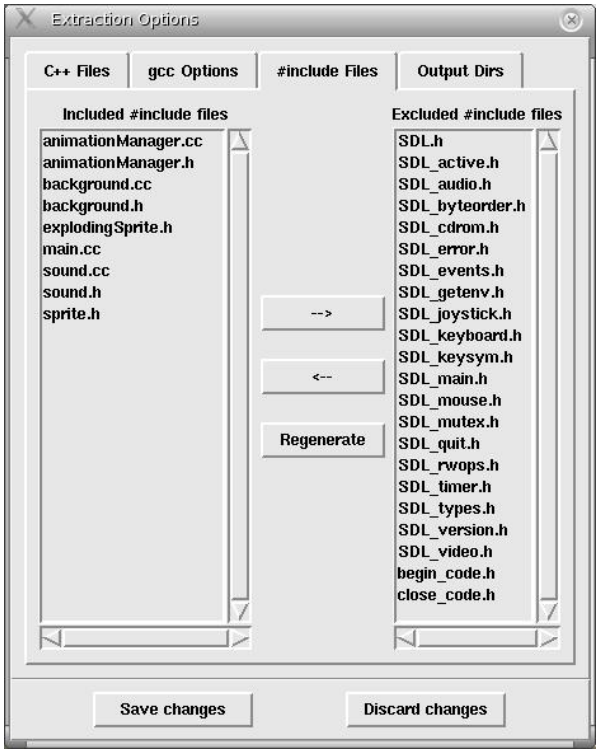

Figure 2: *The Include File Selector.* This figure shows the include file dialog, one of the windows involved in setting extraction options.

and select the methods and method calls to be profiled at run-time. The Selector tool uses *gcc* to parse a C++ project, as described in Section 3, and presents a class diagram and a call graph for the project. The purpose of the Selector is to allow the user to select those method calls in the program that will be visualized at run-time.

The Selector tool allows the user to create a project for each application, setting compiler and other options, and adding the relevant source files to the project. In setting up the project, a user defines the set of C++ files to be processed. Once these have been selected we use *gcc* to calculate file dependencies, and a further dialog box, shown in Figure 2, can then be used to remove individual header files from consideration.

It is not necessary that all the files in an application be added, since the eventual instrumentation code will augment the code base, rather than replacing any elements of it. Being able to select only those files that are of interest is an important aid to comprehensibility, since omitting files will reduce the size of the class diagram and system call graph. Being able to selectively add files to a project also aids modularization, since the source files for an application can be divided over several Selector projects. The output from these projects can then be used either individually or in any combination to instrument the application.

4.1 The Class Diagram

The UML *class diagram* is a popular way of presenting a high-level overview of an object oriented system, and is increasingly used in programming environments as an aid to code organization and navigation. The edges in a class dia-

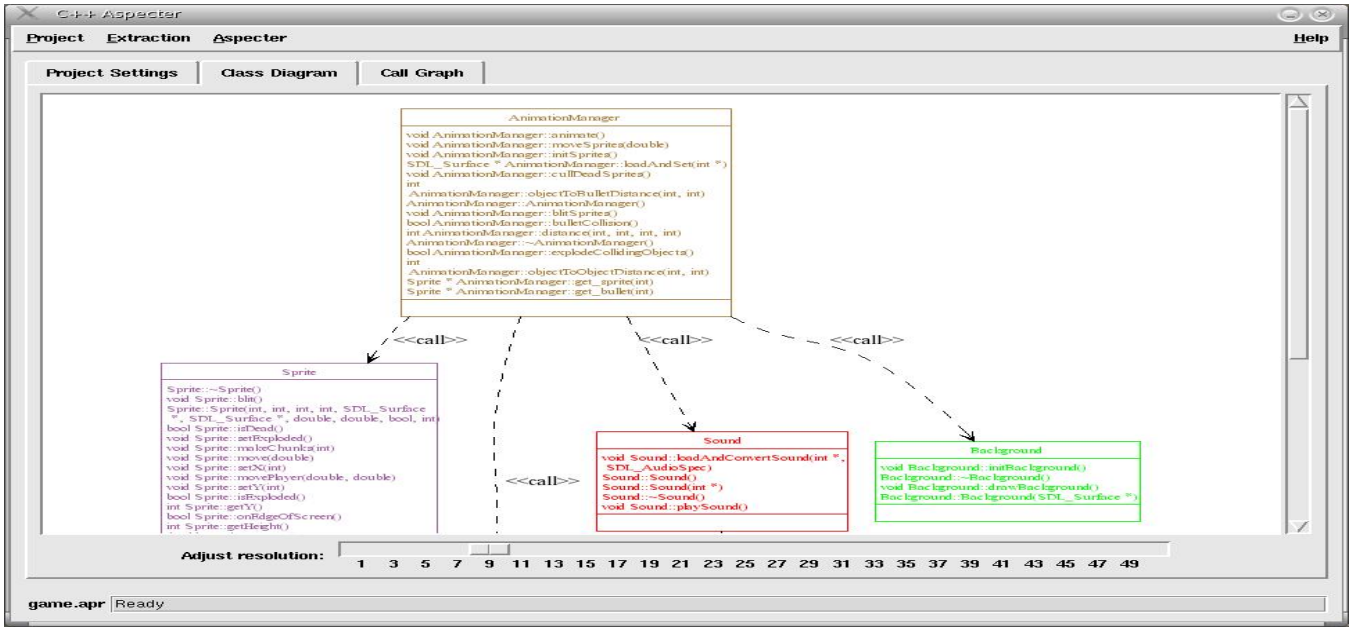

Figure 3: *The Class Diagram.* This figure shows the Selector tool displaying the class diagram for the system. Each class is assigned a unique color, and the layout is based on class dependencies.

gram represent relationships between classes, most notably inheritance. However, associations based on field references, parameters and local variables can also be represented.

Since we are interested in the interactions between methods in the system, we restrict the class diagram edges to just two kinds: those representing inheritance, and those representing dependencies between classes. A class C_1 has a dependency on class C_2 whenever a method from C_1 can call a method from C_2. Eliding associations and using Dot to control the layout gives a rather unusual look to the class diagram. Typically a class diagram, especially when used as a design artifact, will use the inheritance relationship as the main layout ordering edge, and use associations to impose secondary orderings. However, we find that basing the layout on dependencies is a significant aid to understanding the flow of method calls, the main source of flow in our project whose focus is dynamic object modeling. Figure 3 illustrates a partial class diagram for the arcade game that we use as our case study. At the root of the diagram is class AnimationManager, which choreographs the actions of the game. AnimationManager calls methods in classes ExplodingSprite, Sound and Background, with stereotypes attached to each of the edges emanating from AnimationManager to capture this calling relationship.

4.2 The System Call Graph

The second overall view we present is the *system call graph*. This graph is not part of the UML standard, nor does it typically appear during system design, but is a tool commonly used in program analysis and software testing. In a system call graph the nodes are the methods in the system, and there is a directed edge between two methods m_1 and m_2 whenever m_1 can call m_2. Figure 4 illustrates a partial call graph for the arcade game that we use in our case study. In the figure, function main is shown on the left side of the graph with edges leading to the constructor and destructor of AnimationManager, also shown in Figure 3. Figure 4 highlights the fact that all of the actions of the arcade game are directed from AnimationManager.

A theme of our work is that we seek to exploit the widely used UML notations as an aid to program visualization. However, despite the importance of method interactions, there is no UML diagram that completely meets the task of representing these interactions. Elements of the call graph can be derived from UML sequence diagrams, activity diagrams and communication diagrams, yet none of these presents the interactions between methods as clearly as the call graph.

The system call graph presents all methods in the system, and can be quite complicated, even for relatively small C++ programs. An added advantage of our use of Dot to layout the graph is the effect of exposing the ranking between methods in terms of the ordering of method calls, and this provides a good overall view of the hierarchy of interactions. Fixing a coloring scheme for classes and using this coloring for the nodes in the call graph was a significant aid to comprehension of the interactions among the methods in our model.

The system call graph acts as a selector for method calls. We adopt the convention that selecting a node implies profiling all calls to that method, whereas selecting an edge implies only profiling calls from the source method to the destination method. The class diagram acts as a higher level selector, where selecting a class implies the selection of each of its methods in the system call graph.

4.3 Method and Class Call Graphs

One drawback of the high-level view presented by the system call graph is that it can be difficult to distinguish edges for methods that call, or that are called by, many others. To deal with this we have implemented lower-level selectors in both the class diagram and the call graph. Right-clicking on a node in the system call graph presents a detail of the graph, showing just the immediate predecessors and successors of

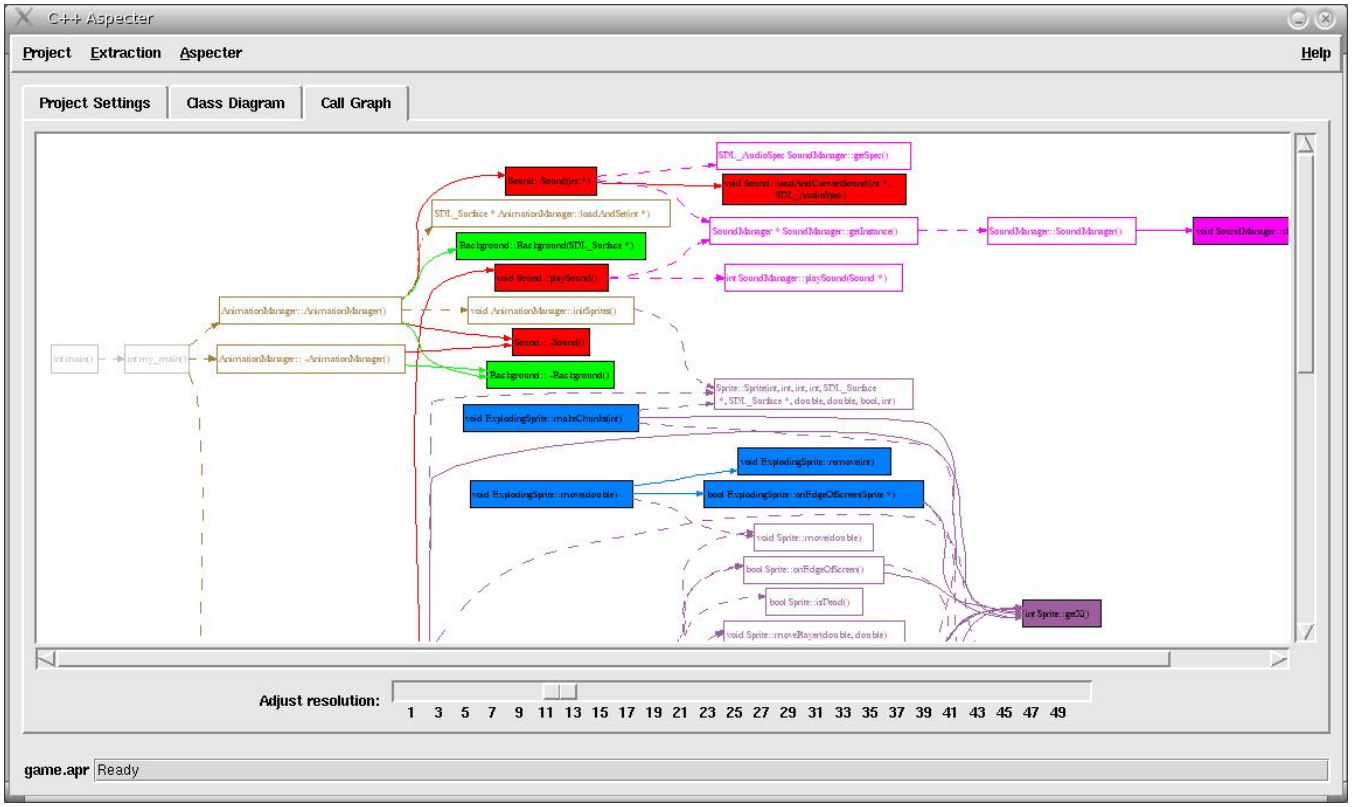

Figure 4: *The System Call Graph.* This figure shows the Selector tool displaying the call graph for the whole system. The nodes in this graph, representing methods, are colored to indicate their owning class.

that node. The reduced information in this *method call graph* can be laid out to clearly distinguish between the method's individual callers and callees. The window at the bottom of Figure 5 illustrates a method call graph where the user has right clicked on method Sound::playSound() and the figure illustrates methods that call Sound::playSound() as well as the methods called by Sound::playSound().

Similarly, clicking the middle mouse button on a class in the class diagram or a method in the call graph produces a *class call graph*. Here, we display the methods in the class, along with the immediate predecessors and successors of these methods from the call graph. These methods are then grouped based on their class, giving a fine grained view on the dependencies between classes. The same coloring and selection conventions are used for all call graphs, and all are kept synchronized, so that a selection in any one is propagated to all the others. The rightmost window in Figure 5 also shows a class call graph where the user has clicked the middle mouse button on a method in class Sound and all of the methods that call methods in Sound are shown as well as the methods called from within class Sound.

Clicking with the middle or right mouse buttons on a class or method call graph replaces that call graph with the new one. Thus the user can display the method call graph for a method, and then follow the chain of calls one-by-one by repeatedly right-clicking on one of the successor methods. At any stage the user can click the middle mouse button on a method and zoom out to the class call graph for that method's class.

At present the Selector provides a single class diagram and system call graph, and allows for any number of popup class or method call graphs. It should be noted however that there is some overhead in maintaining consistency between selections in all these diagrams, and performance can degrade if many views have to be maintained. In practice, we do not envisage a need for more than one or two class or method call graphs at any one time.

5 Dynamic Visualization: The Profiler

The output of the Selector tool is an instrumented version of the original C++ program, which is designed to interact with a simple Python interface at run-time. The Profiler tool described in this section implements this interface, and displays run-time information about a program in terms of UML sequence and communication diagrams.

The design of the Profiler is based loosely on the traditional command-line debugger. The user launches the program from the Profiler, and can then step through its execution at a chosen level of granularity. Unlike a traditional debugger, however, the user does not step through the actual source code, but rather the sequence diagram corresponding to the program's execution. It is not intended that the Profiler would be an alternative to a traditional debugger, but rather a complimentary tool, since it concentrates on visualizing method interactions, rather than the details of method execution.

The main window of the Profiler is shown in Figure 6. The large red 'step' button is used to invoke and return control to the C++ program being profiled, and the slider bar

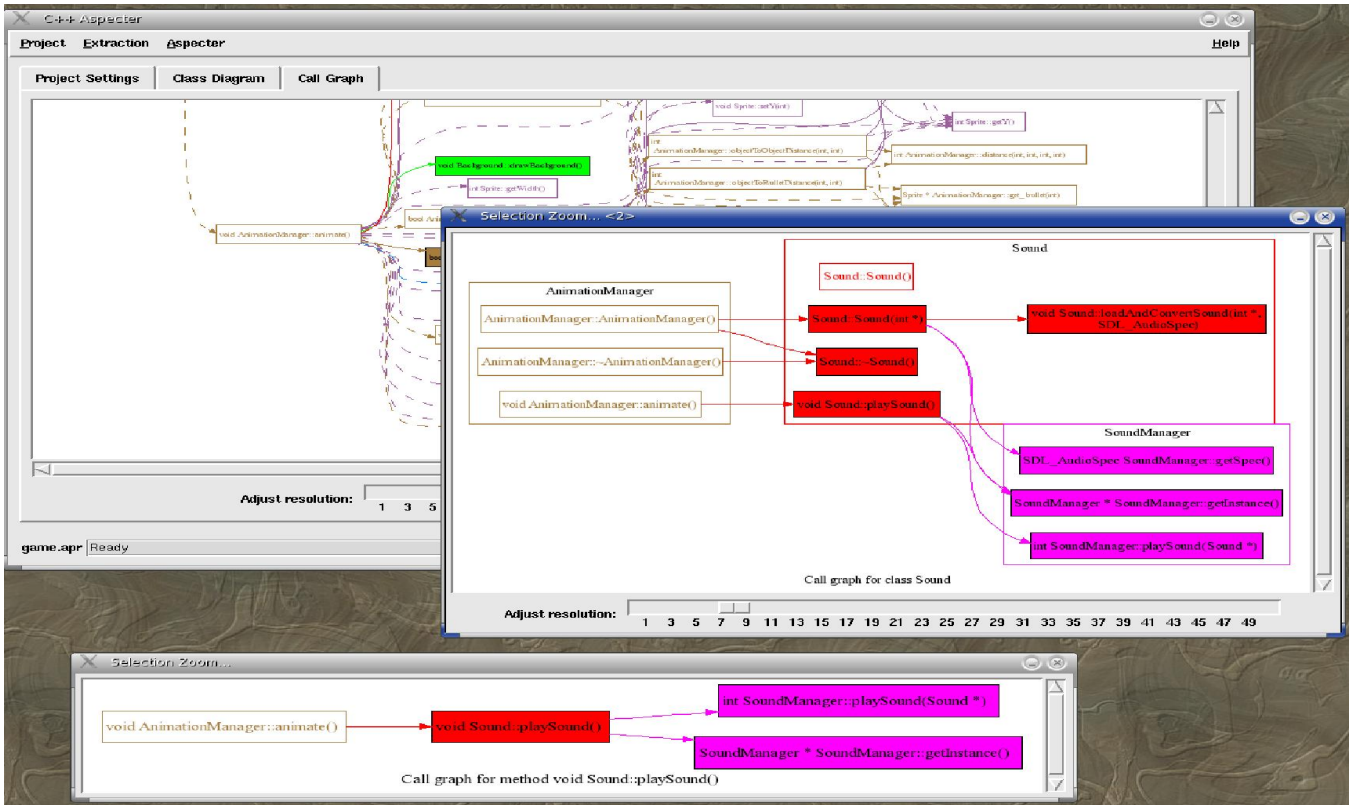

Figure 5: *Class and Method Call Graphs.* These popup windows can be launched by selecting a node in the class diagram or call graph, and show either a class-based or method-based subset of the call graph.

controls the number of events processed between each step. In this context, an event is an object creation or destruction, or the start or end of a method call. The profiler interacts synchronously with the program being profiled. The program is run for the designated number of steps, and then pauses while control is returned to the Profiler so that the user can examine the sequence diagram.

5.1 The Sequence Diagram

The sequence diagram uses the standard UML notation, where each individual object is represented by a horizontal bar, with the object name and class at the top. Each 'thickening' of the bar represents a method belonging to that object being executed. Method calls from one object to the next are represented by horizontal arrows between the relevant object bars. Thus, the sequence diagram increases in width with each object creation, and increases in length with each method call. A large 'X' at the end of an object's line denotes the destruction of that object.

Navigation within the sequence diagram can be achieved using either the slider bars, or the page navigation controls on the panel. As with the selector tool, color is used to identify objects from the same class. The color used in the Profiler is the same as that used in the Selection tool, and is relayed to the Profiler as part of the instrumentation.

Since the sequence diagram can quickly grow quite large, two features were implemented to reduce its width and height. Since a destroyed object has no further use for its horizontal area, newly created objects are always positioned in the leftmost free lane in the diagram. In order to reduce the height of the diagram, the user can choose to ignore any of the methods or objects at run-time. The 'Method filter' button displays a pop-up list of the methods being profiled, and the user can choose to enable or disable tracking of individual methods during the program's run. The 'Object filter' button works analogously for objects.

5.2 The Communication Diagram

The UML communication diagram is a more traditional graph, where the nodes correspond to individual objects, and an edge denotes one or more method calls between the objects. While the communication diagram could be maintained in parallel with the sequence diagram, in practice this imposes a noticeable overhead on the Profiling. Also, the communication diagram quickly becomes incomprehensible for large volumes of data, since it does not follow the more restrictive layout of the sequence diagram, where the most recent events appear at the bottom, and the newest objects appear on the right side of the diagram.

The approach taken in our tool is to provide the communication diagram on request. Thus, whenever the program is stopped and the user is investigating the sequence diagram, they can also press a button to produce the communication diagram for that point in the program. Since communication diagrams can quickly become unmanageable, our dynamic profiler will only present the last 50 events. As is the case for the Selector, an interface to the Dot tool is used to provide graph layout information. Figure 7 shows a profiling session with a communication diagram in the foreground.

We use the standard UML numbering system for com-

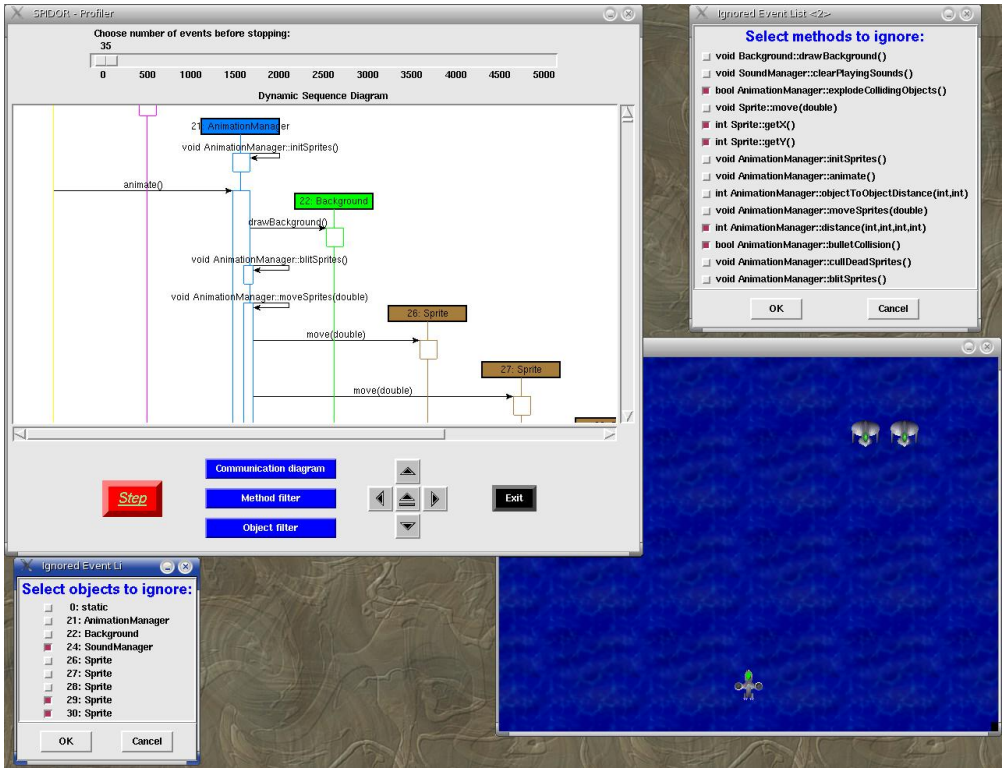

Figure 6: *A Profiling Session.* This figure shows the Profiler tool in action. The Profiler main window, displaying the sequence diagram is shown in the top-left of the figure, and the C++ game being profiled is shown in the bottom-right of the figure. Also shown are the popup method-filter and object-filter dialogs.

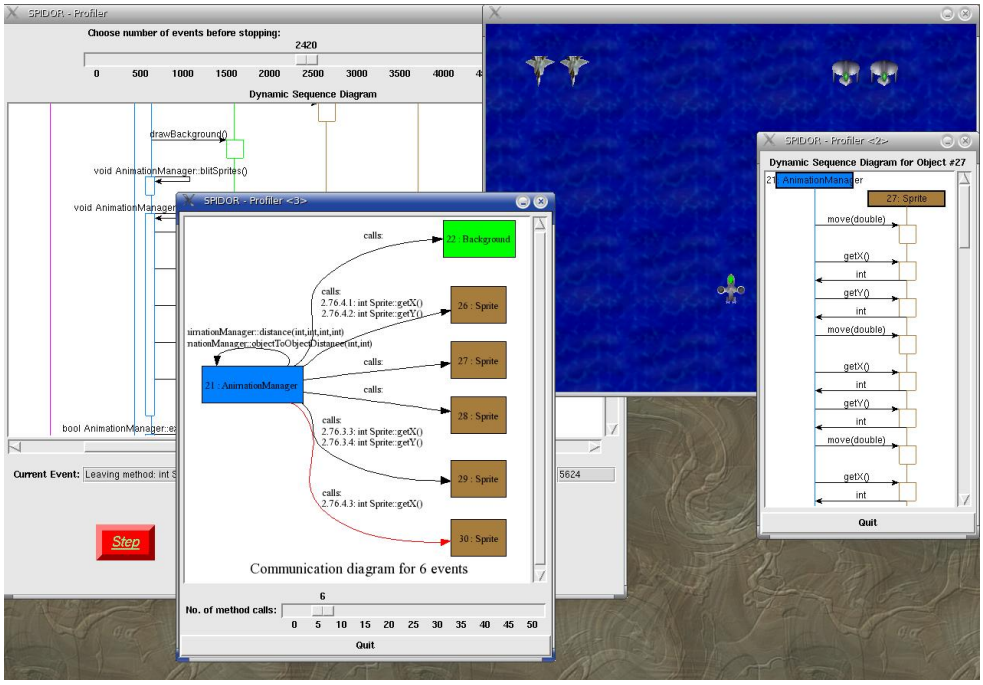

Figure 7: *The Communication Diagram.* The communication diagram, shown here in the foreground, can be launched at any point during profiling. It can show the last 50 events; the slider bar allows the user to step through these events one by one.

munication diagrams in order to identify the sequence of method calls. However, since this can be difficult to follow in larger diagrams, we also provide a facility to mimic the construction of the diagram on an event-by-event basis. The slider bar at the bottom of the diagram allows the user to step through the events one-by-one, and they are filled in on the diagram as the slider bar is moved to the right. The most recently traversed arc and the object owning the currently executing method are highlighted at each step.

5.3 Object-Level Views

Even with static and dynamic event filtering, both the sequence and communication diagrams can rapidly become quite large. The communication diagram in particular can become quite difficult to follow for relatively short sequences of events, especially if these events are concentrated between a few objects. As is the case with our Selector tool, we deal with this problem using selective views of the diagrams.

At any stage during profiling the user can select an object in the sequence diagram and view either a communication diagram or a sequence diagram specific to that object. In both cases we visualize all messages sent to and from the selected object, along with those objects directly involved in those messages. These views allow the user to zoom in on a particular object's activity, without the confusion of sorting through unrelated events. The window on the right of Figure 7 shows a communication diagram for object number 27, an instance of Sprite.

6 Related Work

In this section we review the work that is most closely related to our approach. Due to space constraints we focus on approaches that visualize UML diagrams or demonstrate a filtering approach similar to ours. We note that some of the recent research on garbage collection, for example GCspy [Printezis and Jones 2002], parallels our work but from the perspective of low-level heap visualization rather than UML diagrams.

Jacobs and Musial describe an approach for debugging Java programs using UML object diagrams [Jacobs and Musial 2003]. To capture a large diagram on a single page, they provide more detail for the most recently examined graphical elements and less detail for elements that are remote from this recent element. Their approach enables the viewer to examine as many as 75 classes on a single page, a three-fold improvement over viewing the 24 complete classes that can fit on a page using standard UML presentation such as the presentation provided by ArgoUML.

Jacobs and Musial do not provide static selectors to filter elements for inspection. In our approach, we provide a class diagram and a call graph to permit the user to statically select the relevant parts of the program to be profiled. Moreover, we also provide run-time filtering to enable the user to select the number of events to be viewed, and class and method filtering to allow the user to eliminate selected objects and messages from the profile. Our approach provides a fine-grained view of the interaction of objects and method invocations through our animation of sequence diagrams; we also provide communication diagrams that are symmetric with the sequence diagram currently under view. Finally, our approach allows the user to interact with our profiling system as the application is running, allowing the user to control the filtering and selection in the profile window, transfer control back to the application to obtain further profile information and to iteratively move between the profile window and the application until terminating the run. This interaction with both the profiler and the application is unique to our approach to visualization.

Jones et al. present two tools, TARANTULA and GAMMATELLA, that utilize visualization techniques to effectively transform program execution data into visual information that can be explored and easily understood [Jones et al. 2002; Jones et al. 2004; Orso et al. 2003]. The tools use coloring to summarize information about the program-execution data, using a continuous spectrum of colors ranging from red (danger), to yellow (caution), to green (safety). To manage large programs, they provide representations of the application at three levels of granularity from fine-grained to coarse-grained: statement, file and system. To handle a high number of program-execution data, GAMMATELLA provides filtering, which permits the user to select a subset of executions to be visualized, and summarization, which permits the user to aggregate the program-execution data for a set of executions. Since GAMMATELLA is used for deployed software, filtering can help the user to focus on a subset of the deployed executions and summarization can help the user identify correlations among deployed executions.

The GAMMATELLA system is complementary to our SPIDOR system. Unlike our system, GAMMATELLA does not utilize UML diagrams but rather uses visualization to capture information about the data rather than to promote comprehension of the code through animation of the interactions between objects and method invocations. Also, in our approach the user can dynamically interact with both the profiler and the running application and the user can alter input to the application as it is running; in the GAMMATELLA system the user can only interact with trace data rather than the running program.

Jerding et al. present the Polka animation toolkit, which uses techniques for scalable visualizations based on call trace files generated from C++ source code annotated by hand [Jerding et al. 1997]. Polka visualizations are called *execution murals*, where time is shown on the horizontal axis and each message, visualized as a single pixel wide vertical line from source to destination class, shown on the vertical axis. There are two views provided by Polka. The first view allows a user to display and browse real-world event traces of $100,000+$ messages. The second view attempts to visualize interaction patterns by augmenting the first view with automatic message pattern detection methods. The visualizations of Jerding et al. are not animations of sequence diagrams but rather compactions of interaction diagrams, summarizing hundreds of thousands of messages and the patterns within the messages. Unlike SPIDOR, Polka does not permit interaction with the application during execution.

7 Conclusions and Future Work

In this paper we have described our approach to selecting and visualizing the dynamic interactions among objects in an application. Using the internal abstract syntax tree format of *gcc*, GENERIC, we build representations of a call graph and class diagram to permit the user of our system to statically choose the parts of the program to model. We exploit aspects, using AspectC++, to insert probes into the application to enable profiling of invocations to constructors, destructors and methods. We build a profiler that monitors the dynamic behavior of objects and we visualize this behavior by building sequence and communication diagrams. Since these diagrams can quickly become large, we comple-

ment our static selectors with run-time selectors that permit the user to filter objects and methods from the visualization, thereby reducing the cognitive burden on the user.

The contributions of our work are as follows:

1. The design of a system for dynamic selection and visualization of objects.
2. An examination of the pragmatics of our system through the construction of a toolset, SPIDOR, which reduces the cognitive burden on the user by providing static selection of classes and methods and dynamic selection of objects and messages.
3. The visualization of sequence and communication diagrams illustrating objects and messages of interest to the user.
4. Dynamic interaction with both the application and the profiler. Interaction with the application enables the user to supply input to the application to provide direction and enhance comprehension or debugging. Interaction with the profiler enables filtering of methods and objects for increased cognitive economy.

Our visualization project is ongoing and our future work will take the following directions. We plan to conduct a comparison of the efficacy of sequence diagrams as compared to communication diagrams in reasoning about large C++ applications. To apply our approach to large applications we will investigate techniques to reduce the size of sequence diagrams by collapsing repeating sequences of messages into a single sequence [Jerding et al. 1997]. We will also investigate reducing the size of the communication diagrams using techniques such as those described in [Jacobs and Musial 2003]. We also plan to incorporate more debugging facilities into SPIDOR, permitting the user to set breakpoints during execution of the application. Finally, we plan to investigate usage of SPIDOR generated design artifacts to validate design artifacts constructed earlier in the life cycle.

References

AMBLER, S. W. 2004. *The Object Primer*, third ed. Cambridge University Press.

COOK, S., AND BRODSKY, S. 1999. OMG analysis and design PTF, UML 2.0. In *Request for Information, Response from IBM Corporation.*

GANSNER, E. R., AND NORTH, S. C. 2000. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience 30*, 11 (September), 1203–1233.

GIBBS, T. H., AND MALLOY, B. A. 2003. Weaving aspects into C++ applications for validation of temporal invariants. In *Proceedings of 7th European Conference on Software Maintenance and Reengineering*, 249–258.

JACOBS, T., AND MUSIAL, B. 2003. Interactive visual debugging with UML. In *ACM Symposium on Software Visualization*, 115–122.

JERDING, D. F., STASKO, J. T., AND BALL, T. 1997. Visualizing interactions in program executions. In *International Conference on Software Engineering*, 360–370.

JONES, J. A., HARROLD, M. J., AND ORSO, A. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, 467–477.

JONES, J. A., ORSO, A., AND HARROLD, M. J. 2004. Gammatella: Visualizing program-execution data for deployed software. *Palgrave Macmillan Information Visualization 3*, 3, 173–188.

KICZALES, G., LAMPING, J., MENDHEDAR, A., MAEDA, C., LOPES, C., LOINGTIER, J., AND IRWIN, J. 1997. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, 220–242.

KNAPEN, G., LAGUE, B., DAGENAIS, M., AND MERLO, E. 1999. Parsing C++ despite missing declarations. In *7th International Workshop on Program Comprehension.*

LOHMANN, D., BLASCHKE, G., AND SPINCZYK, O. 2004. Generic advice: On the combination of AOP with generative programming in AspectC++. In *Proceedings of GPCE'04.*

MAHRENHOLZ, D., SPINCZYK, O., AND SCHRDER-PREIKSCHAT, W. 2002. Program instrumentation for debugging and monitoring with aspect c. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 249–256.

MALLOY, B. A., GIBBS, T. H., AND POWER, J. F. 2003. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software, Practice & Experience 33*, 1, 19–39.

MERRILL, J. 2003. Generic and gimple: A new tree representation for entire functions. In *GCC Developers Summit*, 171–180.

MURPHY, G. C., NOTKIN, D., GRISWOLD, W. G., AND LAN, E. S. 1998. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology 7*, 2, 158–191.

ORSO, A., JONES, J., AND HARROLD, M. J. 2003. Visualization of program-execution data for deployed software. In *ACM Symposium on Software Visualization*, 67–76.

PAZERA, E. 2003. *Focus on SDL*, first ed. Premier Press Game Development.

POWER, J. F., AND MALLOY, B. A. 2002. Program annotation in XML: a parser-based approach. In *Working Conference on Reverse Engineering*, 190–198.

PRINTEZIS, T., AND JONES, R. 2002. Gcspy: an adaptable heap visualisation framework. In *OOPSLA*, 343–358.

REISS, S. P. 2003. Visualizing Java in action. In *ACM Symposium on Software Visualization*, 57–65, 210.

ROSSUM, G. V. 2003. *An Introduction to Python*, first ed. Network Theory Ltd, September.

RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. 1999. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley.

SELIC, B. 2004. UML 2.0: Exploiting abstration and automation. *Software Development Times Issue 98* (March 15).