The following paper was originally published in the
Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)
Monterey, California, June 1995.

Program Explorer: A Program Visualizer for C++

Danny B. Lange and Yuichi Nakamura
IBM Research, Tokyo Research Laboratory
Kanagawa-ken 242, JAPAN

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Program Explorer: A Program Visualizer for C++

Danny B. Lange   Yuichi Nakamura

IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato-shi
Kanagawa-ken 242, JAPAN
e-mail: danny@acm.org

Abstract

Despite the obvious advantages of using object-oriented (O-O) program visualizers in system understanding and debugging, they are still rarely found in the programmers's tool box. One reason for this that such visualizers often fail because of their inability to handle problems of a realistic scale. In our research, we have addressed the scalability problem by integrating static and dynamic program information to produce abstract and yet accurate views of complex O-O systems that often provide more useful information than can be obtained by reading the source code. This is the approach we followed in designing Program Explorer, a research prototype for C++ program visualization, which has been used to examine large O-O systems such as Stanford's Interviews library and Taligent's CommonPoint frameworks.

1 Introduction

Whether we want to re-use or debug an object-oriented (O-O) system, we must acquire a thorough understanding of the static and dynamic properties of the system. Although the use of object orientation has changed software development for the better, it has not exactly made programs easier to understand.

Inheritance, **polymorphism**, and **encapsulation** are all good O-O concepts, but also tend to make the actual designs in which they are used more difficult to comprehend. Inheritance makes it difficult to "read" the behavior of a particular object, since that object can belong to a chain of, say, five to ten classes; polymorphism often makes it difficult to determine which method is actually executed in a given object; and encapsulation makes it difficult to understand that no object is an island, but rather that each is a part of a cooperative network of objects.

In fact, understanding O-O systems is the art of combining knowledge about the **concrete** (objects and their interaction) and the **abstract** (classes and their relationships). One cannot fully understand an O-O system by simply considering its concrete or abstract properties in isolation. The former consist of the visible results of the execution of the system, while the latter consist of what we can expect from the system. Understanding evolves from a knowledge of the relations between these two sets of properties.

The process of understanding programs is notoriously difficult and cumbersome. Often many different classes are involved and interaction turns out to be non-trivial. The complexity of both the design and some O-O concepts frequently makes it difficult to verify the scenarios we construct with pen and paper.

Tools such as **CIA++** [9] and **GraphLog** [2] have been developed to help in understanding O-O programs. The problem, however, is that information given by this type of tool for real-world programs is sometimes very difficult to comprehend. There are very few ways of distinguishing relevant information from less relevant information. The root of the problem is that these tools more or less display the obvious: an unfiltered graphical representation of the source code that is just as difficult to comprehend as the source code itself.

Our approach is to combine program execution with the understanding of objects and interactions, and the static program information (source code) with the understanding of classes and their relationships. This allows us to determine which classes are relevant and how program behavior changes according to our interaction with the running program.

The challenge of this approach is that dynamic analysis of O-O systems involves generating huge amounts of program information that is hard to digest, especially if the information is presented in a purely textual format. The situation is not unique. Other scientific areas characterized by huge amounts of data face the same problem. One solution that has become increasingly popular is called scientific visualization. It has in many cases proven to be a particularly good way of presenting a large amount of information [16].

Two O-O program visualization tools that utilize dynamic information are **Object Visualizer** [17, 18] and **HotWire** [12]. Both rely on visual effects to draw attention to program anomalies, rather than giving exact information. Although this approach appears efficient for detecting problems and, to some extent, for localizing them, it hardly provides the exactness that programmers need in order to understand a problem and correct it.

In our research, we have developed a mechanism capable of amplifying patterns of object interaction in program visualizations. Our approach is based on the observation that static information can leverage dynamic information and vice versa. The technique is to let the dynamic information impose (1) a sense of relevance on the static entities, such as which classes or member functions are actually important during a certain phase of execution, and (2) a sense of sequence, such as the order in which member functions are called. In contrast, static information allows us to focus on (1) objects of certain classes, and (2) properties related to certain classes.

Hence, the two main issues that have to be addressed in order to visualize O-O programs are the availability of program information (capturing static as well as dynamic information), and the scalability of information processing and presentation (that is, the ability to create useful visualizations of real-world systems). We have addressed both these issues in the development of a research prototype for program visualization named **Program Explorer**.

Program Explorer is a system for understanding C++ programs through visualization. Its purpose is to provide class- and object-centered views of the structure and behavior of large C++ systems, with information accurate enough to enable programmers to re-use and maintain undocumented parts of such systems. The tool should be graphically oriented and should provide interactive hypertext-like navigation of program entities in running programs.

In the following section, Program Explorer's way of coupling static and dynamic program information is described and some visualizations are displayed. Section 3 presents the **Program Database**, the source of static program information. Sections 4 and 5 describe how the dynamic program information is generated and retrieved. Section 6 presents our conclusions.

2 Information Coupling and Visualization

From the Program Database, Program Explorer can retrieve static program information for a number of interesting visualizations such as class inheritance hierarchy, function calls, and variable access. From the execution trace of a running system it can generate a number of vi-

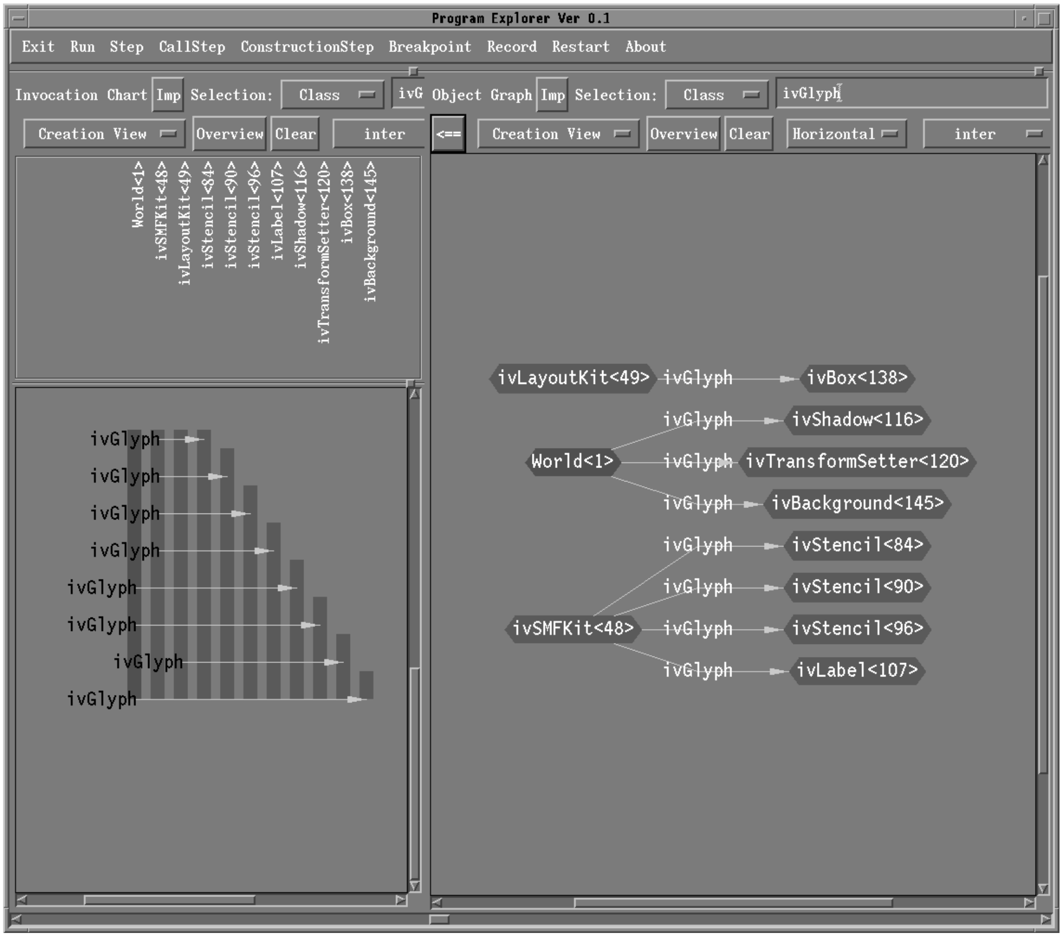

Figure 1: *Class-to-Object Coupling: Vertical Selection.*

sualizations that can give insights into its dynamic properties, such as object creational hierarchy, object calls, and object usage. It is thus possible to investigate the exact sequences of interaction among several objects needed to accomplish some task, and to detect program anomalies such as objects being created but never destroyed, and objects being invoked after they have been destroyed.

However, in our experience the main problem with pure static and pure dynamic models is their limited applicability to systems of a realistic size. The reason for this is that, in a sense, pure static views as well as pure execution views just show the obvious. That is, they show what is already present in the source code and in the computer memory during program execution. What helps in program understanding is the coupling of the abstract and the concrete, that is, of static and dynamic information. In Program Explorer we have found two coupling techniques that are particular useful, namely, **Classes-to-Objects** and **Objects-to-Classes**.

The Classes-to-Objects coupling can be used to filter dynamic information by means of static information. In Program Explorer, we have defined two particular categories of selection based on the class inheritance graph: **vertical** and **horizontal selection**. By vertical selection we mean the process of selecting objects of a given class. By horizontal selection we mean the se-

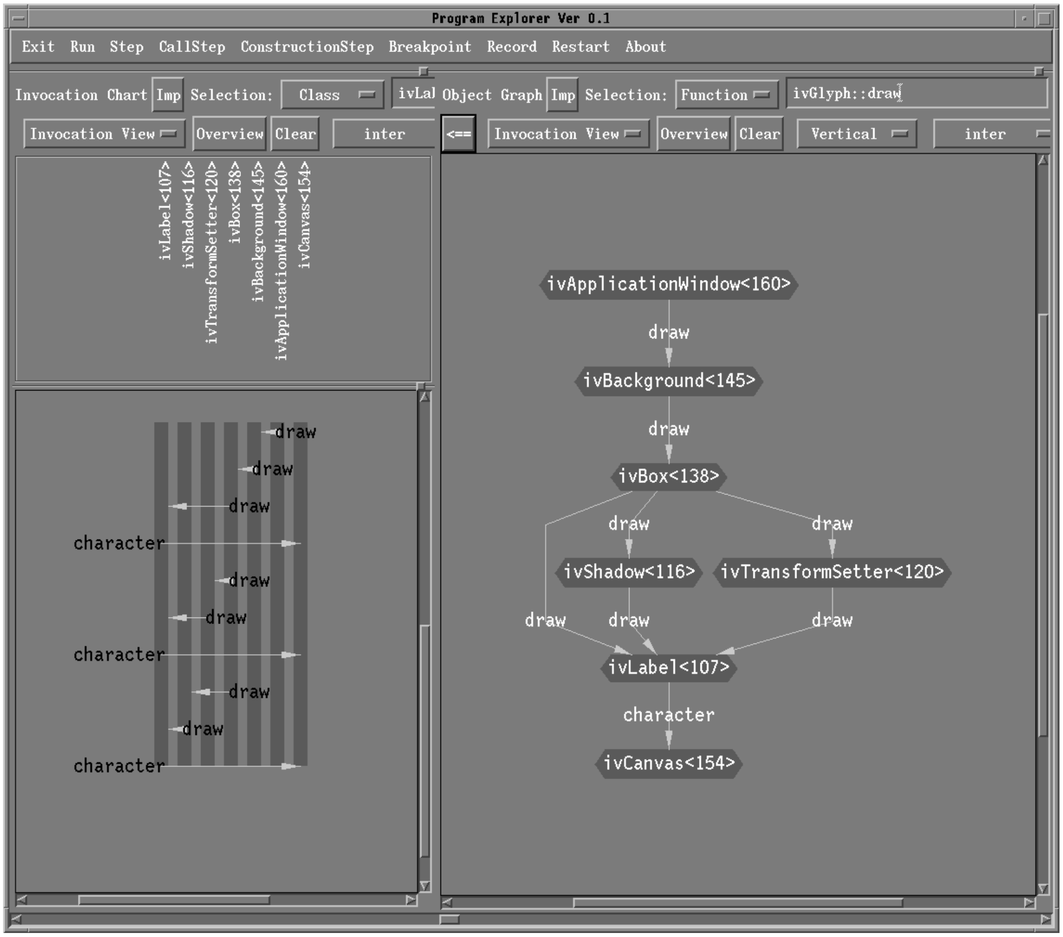

Figure 2: *Class-to-Object Coupling: Horizontal Selection.*

lection of certain object properties related to a specific class. Often, we use vertical selection to select a group of objects, and then use horizontal selection to study particular aspects of their interaction.

We have created an Interviews [14] sample program to examine the use of flyweight[1] objects in Interviews (see Appendix A and Figure 7). We know that `ivGlyph` is the root class for all graphical objects; thus, by retrieving objects derived from this class we can examine all graphical objects produced by this sample program. Figure 1 demonstrates such use of vertical selection. In the right pane we see the **Object Graph** of graphical objects and their creators. The nodes in the graph are objects identified by their class name and a unique number (`World<1>` represents the un-instrumented creators). The arcs represent creational relationships. In the left pane we find the **Invocation Chart**, which displays object longevity (the lengths of the bars) and the order of creation (from top to bottom).

Following the vertical selection, in Figure 2 we show an example of horizontal selection. The class `ivGlyph` defines a virtual function `draw` that is implemented in each of the derived classes. By selecting `ivGlyph::draw` we focus on a single behavioral aspect of these objects. The hierarchical drawing process from window

[1]Flyweight objects are fine-grained objects that can be shared efficiently.

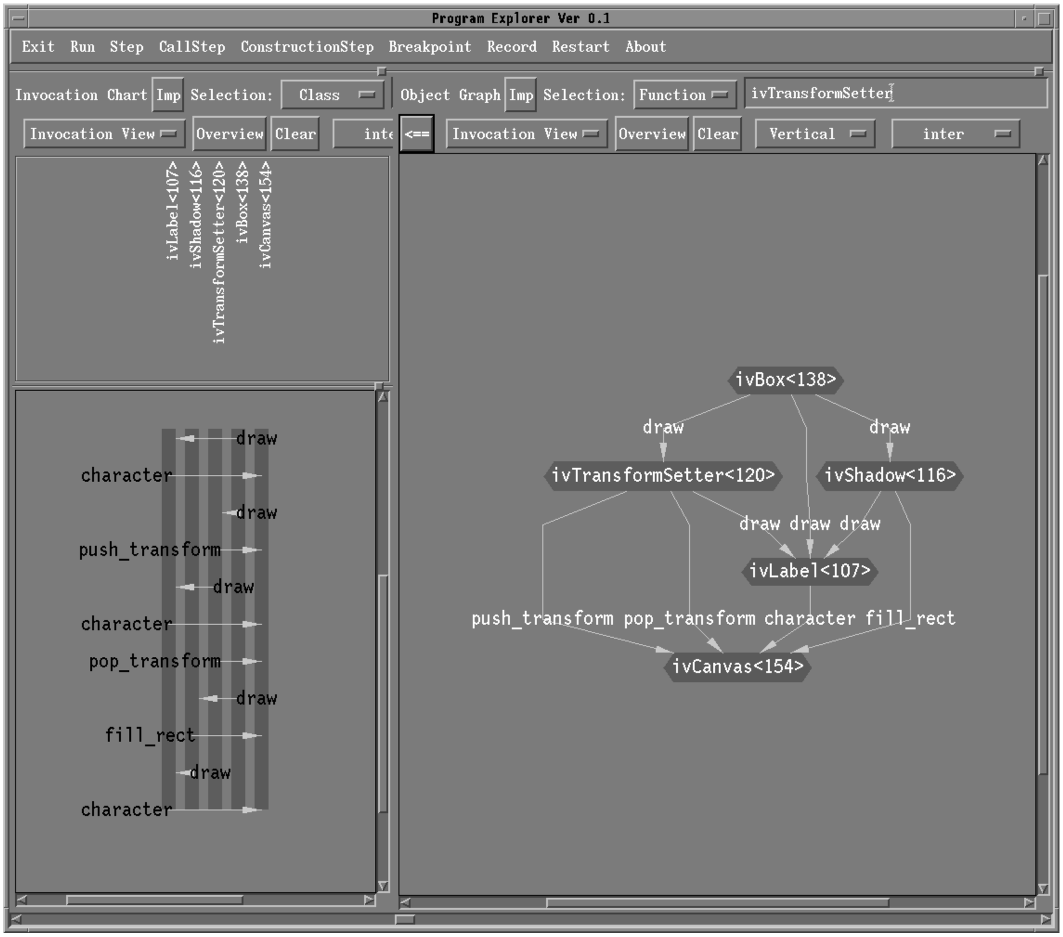

Figure 3: *Moving Focus to* `ivTransformSetter`.

to canvas becomes very clear, particularly in the invocation chart. It also shows how the label, as a flyweight object, is reused and drawn in three different contexts: plain, with a shadow, and transformed (rotated).

By Objects-to-Classes coupling we mean the transfer of dynamic information to the static domain. The purpose of this coupling is to filter huge amounts of static information in such a way that only the relationships actually used are in focus. Such visualizations can be used to express class-proximity (quantification of dynamic information to determine how dependent classes are on each other), or they can be used in class-based diagrams to describe object communication by numbered call relationships among classes, indicating the order in which invocations are supposed to take place. Surprisingly informative visualizations can be obtained by applying static and dynamic information coupling to large systems.

Orthogonal to the static and dynamic representations of O-O program executions are their **visual representations**. The purpose of a visual representation is to communicate the content of a given view of an O-O program and its execution. Different visual representations have different characteristics that make them more or less suited for particular views. In our research we have implemented and studied a number of interesting visual representations including graphs, bar charts, and matrices. The use of

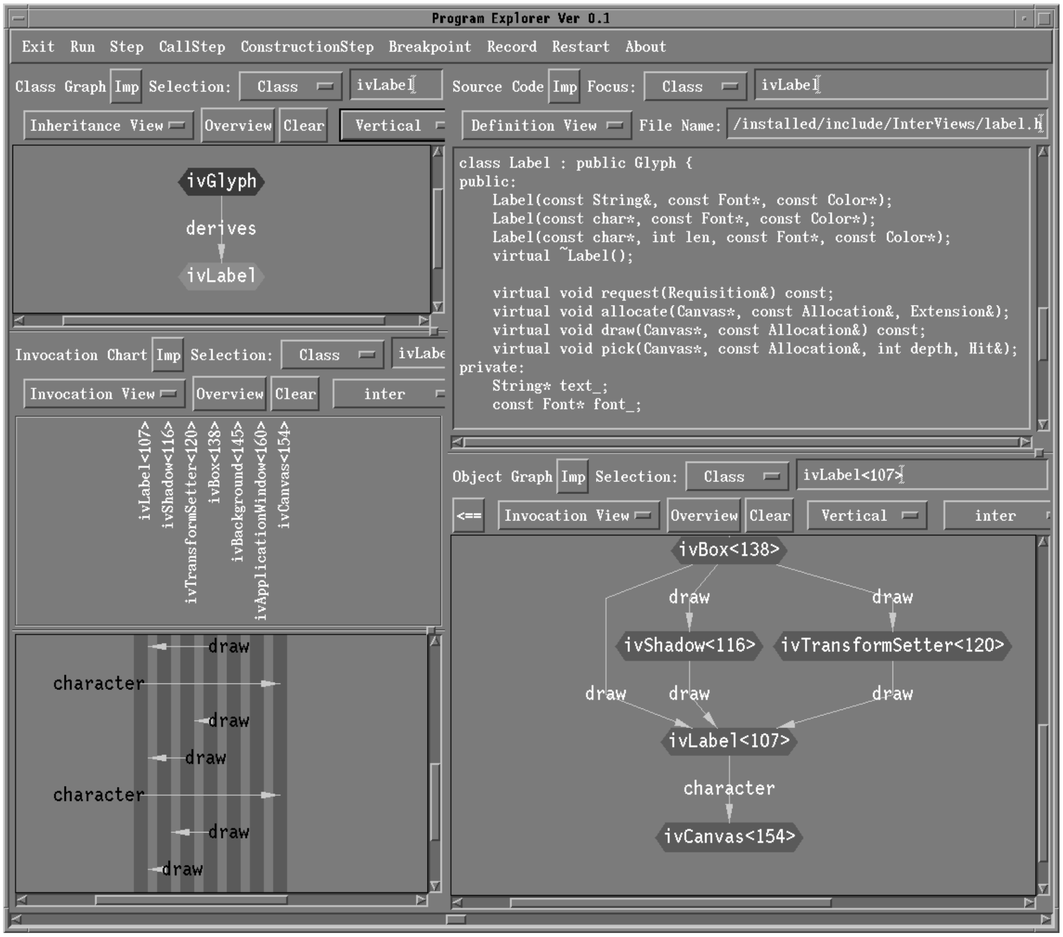

Figure 4: *View Integration: Four Views of* `ivLabel`*.*

color has also been explored. Our color convention (which can be re-defined by the user) is red for objects, green for classes, blue for free functions, and brown for the un-instrumented part of the system. Highlighting is used to display selected entities.

Two interaction techniques are used in Program Explorer's GUI to deal with the issue of scalability. **Navigation** is the technique used within a given view to explore it in an step-by-step fashion comparable to that of hypertext linking. In the previous example, we can move the focus to the `ivShadow` and `ivTransformSetter` objects to examine how a label is actually given a shadow and how it is rotated. In Figure 3 we have moved the focus to one of the above objects, and we can see how `ivTransformSetter` actually instructs the canvas to rotate the label object without the latter's knowledge. Navigation allows the user to focus on certain parts of a graph while ignoring others.

A focus can also be exported to another view. As is shown in Figure 4, Program Explorer's GUI consists of four panes, of which three are graphical and one is textual. The **integration** of these four panes through the focusing mechanism enables the user to view static properties in one pane and dynamic properties in another. We regard this as a GUI-based coupling of static and dynamic information.

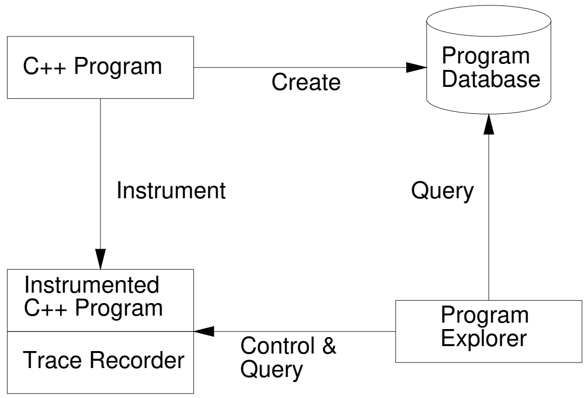

Figure 5: *The System Architecture of Program Explorer.*

3 Static Program Information

Retrieving static program information for C++ programs is strictly speaking not a part of Program Explorer. It is retrieved by the **Program Database** from so-called `pdb`-files generated by IBM's **xlC** compiler.

An overview of the system architecture of Program Explorer can be seen in Figure 5. The system includes a program database for C++, an **instrumentation utility** for augmenting C++ programs with code that produces trace information, a **Trace Recorder** linked to the instrumented programs that captures the trace, and finally, Program Explorer, which controls program execution and presents static and dynamic information through its GUI.

The Program Database is a stand-alone application that implements the schema of static program information from Table 1 and provides a full Prolog interface to clients. A simple query

```
pd_class(Cid, Cname, class,_,_)?
```

will return a set of pairs of (*ClassID*, *ClassName*). The query can be extended to

```
pd_inherit(DerID, BaseID,_,_,_,_),
pd_class(DerID, DerName, class,_,_),
pd_class(BaseID, baseName, class,_,_)?
```

which returns the set of binary inheritance relationships between classes. The built-in Prolog interpreter can also compute recursive queries, and the uniform representation of facts and rules in Prolog allows clients to append rules to the database and thus create their own logic programs as a part of the database.

4 Program Instrumentation

Collecting accurate information on a running C++ program is not easy. Without a meta-class concept, there are basically three ways of collecting such information: (1) extend the compiler so that it adds trace-generating code to the normal code, (2) use debugging techniques, or (3) augment the source code with trace-generating code.

Many compilers offer options for generating profiling information that can be used to detect "hot spots" in the code. Traces of this type do not provide enough information for our kind of visualization. Ideally, compilers can be modified to insert code that produces detailed trace information, but we considered that task to be beyond the scope of our work.

Another way of retrieving dynamic information is to use debugging techniques. Normally, this is done by setting breakpoints at the functions in a program. However, when dynamic information is collected in this way, a great deal of time is spent by the system in context switching between the process of the target program and the process of Program Explorer. Processes are heavyweight processes in AIX and a context switch from process to another is a costly operation.

The third solution is to augment the source code with trace-generating code. This technique is often termed **program instrumentation**. C++ preprocessors have been used to instrument programs [17], but a complete instrumentation would require a semantic analysis comparable to the one carried out in the compiler. We found instrumentation to be an

Table 1: *Schema for Static Information.*

- **Facts on entities:**

```
pd_directory(ID, Name, PathName, ParentDirID)
pd_file(ID, Name, PathName, Time, Language, DirID)
pd_class(ID, Name, ClassType, IsSOM, SOMName)
pd_function(ID, Name, PrototypeString, FuncStorageClass, Const, Inline, Overload,
             Operator, VirtualSpecifier, FuncMiscAttriute, Volatile, IsSOM, SOMName)
pd_variable(ID, Name, DeclarationString, VarStorageClass, Const, Volatile)
pd_enumeration_tag(ID, Name)
pd_enumerator(ID, Name, EnumID)
pd_macro(ID, Name, NumberOfArguments)
pd_typedef(ID, Name, DeclarationString)
pd_label(ID, Name)
pd_template(ID, Name, TemplateType, LongName)
```

- **Facts on relationships:**

```
pd_defined(ID, ScopeID, FileID, Line, Column)
pd_declared(ID, ScopeID, FileID, Line, Column)
pd_used(ID, ScopeID, FileID, Line, Column)
pd_used_implicit(ID, ScopeID, FileID, Line, Column)
pd_used_lvalue(ID, ScopeID, FileID, Line, Column)
pd_member(ClassID, MemberID, AccessSpecifier, Offset)
pd_friend(ClassID, FriendID, Line, Column, FileID)
pd_inherit(DerivedClassID, BaseClassID, AccessSpecifier, Virtual, Order, FileID)
pd_include(FileID, IncludedFileID, Line)
pd_instantiated(InstantiatedID, TemplateID)
pd_source2pdb(SourceFileID, PdbFileID, CompilerOption, Language)
pd_call(CallerID, CalleeID)
```

- **Facts on types of names:**

```
pd_typeof_variable(ID, Type)
pd_typeof_function(ID, ReturnType, NumberOfArguments, ListOfArgumentTypes)
pd_typeof_typedef(ID, Type)
```

acceptable solution, but instead of using a C++ preprocessor we decided to rely on the Program Database for accurate instrumentation information. The contents of the Program Database are compiler-generated and thus very exact with respect to the semantics of program entities and their physical location.

Instrumentation. Program Explorer provides selective instrumentation on a class-wise basis. The GUI of Program Explorer allows the user to specify which classes should be instrumented. A directory-file-class hierarchy allows flexible selection of whole directories, files, and classes. This technique is suitable for avoiding trace information from highly active classes and from classes that are trivial or already well understood. We intend to extend this technique to encompass functions as well as variables.

Our aim has been to instrument C++ programs to produce complete and accurate trace information. For that purpose it is necessary to capture events related to **object longevity**, **function invocation**, and **variable access**. Program events are captured by the Trace Recorder through the internal protocol (see Figure 6 and Table 2). The Trace Recorder processes events to produce a trace, which it also stores. Program execution is controlled and trace information is queried by Program Explorer through the Trace Recorder's external interface.

It should be emphasized that the instrumentation code shown below is in no way specific to IBM's `xlC` compiler. The code fragments are generally applicable for tracing C++ programs, and only the way in which we insert those frag-

Table 2: *Internal Protocol for the Trace Recorder.*

	Command	**Arguments**			
Longevity	`Allocate`	*ObjectID*	*ClassID*	*MemoryAddr*	
	`Deallocate`	*ObjectID*			
Invocation	`Construct`	*ObjectID*	*ClassID*	*FunctionID*	
	`Destruct`	*ObjectID*	*ClassID*	*FunctionID*	
	`Enter`	*ObjectID*	*ClassID*	*FunctionID*	
	`Leave`	*ObjectID*	*ClassID*	*FunctionID*	
Access	`Usage`	*ObjectID*	*VariableID*	*RetrFunc*	*ObjectAddr*

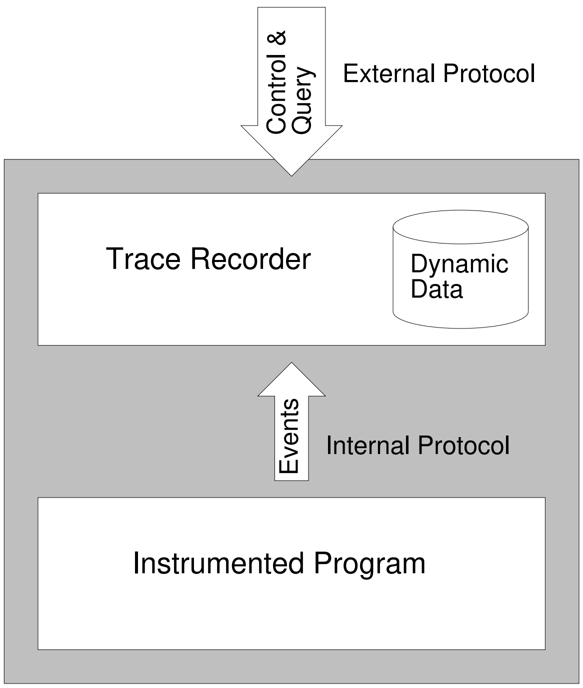

Figure 6: *The Executable: Program and Trace Recorder.*

ments is specific to the `xlC` compiler and the Program Database.

Object Longevity. Two events related to object longevity are captured: creation and destruction of an object. For this purpose every instrumented class is equipped with a "identity object", `_PEoid`:

```
class A {
   PE_Oid _PEoid;
...
}
```

For this instrumentation we use C++'s automatic construction and destruction of class members. The constructor of `PE_Oid` assigns a unique number to the object and calls `Allocate` in the Trace Recorder, whereas the `PE_Oid`'s destructor calls `Deallocate`. The constructor and destructor functions of `A` are not suitable for capturing object creation and destruction, since such events actually take place respectively before and after these functions are called.

Where should we put `_PEoid` in the case of inheritance? We put it into each instrumented class, even if these classes appear in the same inheritance path. To avoid allowing one object to have multiple identities, subclass `_PEoids` "inherit" their unique identity (an integer value) from the base class. In the case of multiple inheritance this mechanism works the opposite way.

```
class B : public A {
   PE_Oid _PEoid(A::_PEoid.oid());
...
}
```

Why not use the value of `this` as a unique id of an object? This would not work, for several reasons. First, if multiple inheritance, possibly with virtual base classes, is involved, different parts of an object (depending on which class in the inheritance path that part is representing) will return different values of `this`. Second, when an object is deallocated and a new object is created in the same space, it would be difficult to maintain a unique trace as object ids would no longer be unique over time.

Function Invocation. Function invocations are captured with a mechanism identical to the one described for object longevity. `_PEtmp` is a local class variable that is automatically constructed when the function is invoked, and destructed upon its return:

```
A::F() {
    PE_Func _PEtmp(_PEoid.oid(),
                   ClassID,
                   FunctionID);
...
}
```

The constructor of `PE_func` calls `Enter` in the Trace Recorder, and the destructor calls `leave`. One of the benefits of this approach is that only the callee is instrumented, and since one function may have many callers, it more efficient than instrumenting call locations.

The C++ compiler silently generates a number of functions if they are not explicitly defined by the programmer. Such functions are the *constructor*, *destructor*, *copy-constructor*, and *assignment operator*. For more details on this subject, consult the C++ ARM [5].

Regardless of the implicitness of these functions, they often play an important role in the execution of an O-O program and thus cannot be ignored. The copy-constructor creates new objects, and along with the assignment operator it copies the state of one object to another. No source code exists for compiler-generated functions, so we need to make these functions explicit.

Below is an example of a constructor and destructor. Notice that `PE_Constr` and `PE_Destr` are used instead of `PE_Func` to distinguish these three types of function invocation. The constructor of `PE_Constr` calls `Construct` in the Trace Recorder and `PE_Destr` calls `Destruct`. The constructor instrumentation is as follows:

```
A::A() {
    PE_Constr _PEtmp(_PEoid.oid(),
                     ClassID,
                     FunctionID);
...
}
```

and the destructor instrumentation is as follows:

```
A::~A() {
    PE_Destr _PEtmp(_PEoid.oid(),
                    ClassID,
                    FunctionID);
...
}
```

The copy-constructor and the assignment operator are more complicated to create than the above constructor and destructor. In both cases it is necessary to create separate initialization lists for scalar member variables and assignment routines for array members. Below is an example of an explicit copy-constructor. Notice that `_PEoid()` in the initializer list ensures that the copied object receives a new identity different from the originator (`rhs`).

```
A::A(A& rhs) : _PEoid(),
               r(rhs.r), s(rhs.s),...
{
    PE_Constr _PEtmp(_PEoid.oid(),
                     ClassID,
                     FunctionID);
    Array member assignment
}
```

Free functions (non-member functions) are also instrumented. In their case the class identity and object identity are set to zero.

Variable Access. Variable access is more difficult to capture than function invocations. The

Table 3: *Schema for Dynamic Information.*

- **Object longevity:**

```
create(SrcObjID, SrcFunID, TgtObjID, ClassID, Time)
destroy(SrcObjID, SrcFunID, TgtObjID, Time)
```

- **Interactions:**

```
invoke(SrcObjID, SrcFunID, TgtObjID, TgtFunID, Time)
access(SrcObjID, SrcFunID, TgtObjID, VariableID, Time)
```

- **State:**

```
value(VariableID, Value, Time)
```

approach we have taken is to "functionalize" variable access. That is, each definition of a member variable is attributed by an access function to be used instead of direct variable access. The original member definition

```
B* b;
```

is supplemented by two functions. The first serves to capture variable access:

```
B*& b_PE() {
    PE_usage(_PEoid.oid(),
             VariableID,
             A::b_PEvalue);
    return b;
}
```

while the second is used to retrieve the value of the variable:

```
static void* b_PEvalue(void* o) {
    return (void*)((A*)o)->b;
}
```

The access function (`PE_usage(...)`) notifies the Trace Recorder about variable uses. It also forwards a pointer to a function able to return the value of the variable. The reason the Trace Recorder does not receive the value directly is that variable assignment first takes place after the access function has returned. Member variable access is modified by adding the function-suffix `_PE()`. This suffix works for both reading a variable:

`b->foo();` *becomes* `b_PE()->foo();`

and writing to a variable:

`b = new B;` *becomes* `b_PE() = new B;`

5 Trace Recording and Execution Control

When an instrumented program has been compiled and linked with the Trace Recorder, it is ready to be executed by Program Explorer. Below we describe trace recording and how it is controlled and used by Program Explorer.

Trace Recording. The atomic events of a running program are captured by the Trace Recorder and transformed into a sequence of binary relations. The event in which an object is created is transformed into a relation that specifies the identity of the object being created as well as that of its creator. Function invocations are captured as a sequence of `Enter`- and `Leave`-events that can easily be converted into the corresponding series of binary call relationships. Binary relations are more convenient than raw events with regard to storage management and query processing. The schema for representing these relationships is given in Table 3.

Since the Trace Recorder stores and manages the trace, Program Explorer has to pose queries

Table 4: *External Protocol for the Trace Recorder.*

	Command	**Input**	**Output**
Execution Control	`exit`		
	`run`		*Focus Record*
	`step`		*Focus Record*
	`callStep`		*Focus Record*
	`returnStep`		*Focus Record*
	`constructionStep`		*Focus Record*
	`usageStep`		*Focus Record*
	`setBreakPoint`	*Focus Record*	
Trace Recording	`invocRecording`		
	`usageRecording`		
Query Processing	`about`		*Result*
	`getInvocations`	*Focus Record*	*Result*
	`getConstructions`	*Focus Record*	*Result*
	`getUsages`	*Focus Record*	*Result*
	`getPointers`	*Focus Record*	*Result*

to the Trace Recorder in order to produce visualizations of dynamic information. The query interface to the Trace Recorder is a part of its external protocol given in Table 4. Unlike the Program Database, the Trace Recorder only provides a fixed number of query functions. This restriction has been caused by a requirement of low response times and space-efficient storage of large traces. Still, the query mechanism is very flexible, since each query can take a **Focus Record** as argument. A Focus Record specifies any meaningful selection of a static or dynamic program entity or relationship, such as *class*, *object*, *function*, or *invocation*.

Queries take the form of

`someQuery(`*ClassID*`,`
SrcObjID`,` *TgtObjID*`,`
FuncID`/`*VarID*`)`

and return lists of

`Result(`*SrcClassID*`,` *SrcObjID*`,`
TgtClassID`,` *TgtObjID*`,`
FuncID`/`*VarID*`)`

Notice that, since no text information is exchanged between Program Explorer and the Trace Recorder, the Program Database acts as a name server. Using unique class, function, and variable identifiers gives a very compact trace, lowers the communication overhead, and allows the system to distinguish overloaded names.

Execution Control. The instrumented program is executed under the control of Program Explorer. That is, Program Explorer uses the control interface of the Trace Recorder's external protocol to run the instrumented program or to execute it in a single-step mode. This mechanism is well known from debuggers and very suitable for localization (finding a spot of particular interest). Whenever execution in the instrumented program is halted (that is, when the program exits, or reaches a breakpoint, or when a signal from the operating system is received), a Focus Record is returned to Program Explorer, allowing it to retrieve information about the events that led to that halt. Unlike in debuggers, this information includes full details of all the recorded events up to the halt, and not only the contents of the call stack.

The Trace Recorder also allows selective trace recording. Both invocation and variable usage recording can be switched on and off independently, thus limiting the generation of trace information. This technique and the selective instrumentation mentioned in Section 4 are the two means of reducing trace generation. Selective instrumentation is limited to compile-time. In the future we would like to extend this mechanism to run-time, so that selective trace recording is supported by two orthogonal concepts: program-entity-based and breakpoint-based selection.

Statistics. Table 5 shows statistics from the trace recording of an instrumented version of the Interviews library. For some common Interviews sample and application programs we show the number of different class and function definitions involved in a particular execution, and the number of actual object creations and function invocations. In these examples, no attempts were made to limit trace generation. An important observation that can be drawn from this table, is that, while the amount of dynamic information grows rapidly in proportion to the size and complexity of the application program, the size of the static information space remains almost constant. This observation supports our approach of using static information to leverage dynamic information.

6 Related Work

Two query-based program visualization tools — **CIA++** [9], developed at AT&T Bell Laboratories, and **GraphLog** [3], developed at the University of Toronto — focus on the static properties of O-O systems. CIA++ builds a relational database of information extracted from C++ programs. The database serves as a foundation for static analysis tools for displaying various views of the program structure. GraphLog is a visual tool for databases. Queries are posed by drawing graph patterns with a graphical editor. GraphLog has been used for visualizing and querying software structures [2]. Since both tools are limited to static program information, their visualizations are of limited interest unless very interesting[2] queries are posed. In our experience, however, writing such queries is often difficult and distracts the user's attention from the original goal of understanding a program.

Object Visualizer [17, 18] and **HotWire** [12], both developed at the IBM T. J. Watson Research Center, are dynamic O-O program analyzers that primarily rely on visual effects to draw attention to program anomalies rather than giving exact information. Both tools are based on the same program instrumentation mechanism for gathering execution information. This mechanism is seemingly less accurate than Program Explorer's, and does not generate information on implicit[3] functions, variable usages, and variable values. The execution model of Object Visualizer [18] is accumulative, and is intended for O-O profiling uses such as finding "hot spots" in classes and objects. Another capability of this tool is visualizing "memory leaks," that is, objects that are not deleted after use. HotWire is a visual debugger for C++ that allows the user to write custom visualizations in a scripting language. While this scripting mechanism seems to be very useful for algorithm and object animation, we fear that it distracts the user's attention from program debugging (which often is performed under strong time pressure). HotWire resembles Program Explorer more closely than does Object Visualizer. HotWire and Program Explorer support microscopic views into the state and behavior of individual objects, whereas Object Visualizer focuses on the overall picture characteristic of program profiling tools.

To our knowledge, the systems described in the above have been mainly applied to programs written in C++. However, visualizers have also

[2] Such as whether a base class directly or indirectly relies on properties of one of its derived classes (visualizing bad programming style).

[3] A number of C++ member functions are silently generated by the compiler.

Table 5: *Trace Recording Statistics.*

Application Program	himom	flyweight	preview	idemo	doc
LOC	29	31	197	635	15,210
Classes	42	53	84	112	85
Functions	263	365	514	731	564
Objects	152	271	9,893	9,437	17,404
Invocations	603	1,146	32,327	33,291	64,553

been constructed for other O-O languages such as Smalltalk (e.g., message diagraming [4], the Trick system [1], and Portia [8]), and LISP dialects (e.g., GraphTrace [11]). A common feature of these systems is that they benefit from the openness of interpreted O-O languages. Objects actually exist at runtime in these systems, whereas runtime structures in C++ are flat and not very O-O.

7 Conclusion

Scalability has been the major issue in the design and implementation of Program Explorer. The issue is complex, since it concerns human as well as computer resources. However, we have in our research found a common denominator for addressing this issue: static-dynamic information coupling.

Let us take the computer resources first. With a practically complete instrumentation utility, Program Explorer has been successfully used to instrument and generate trace information for large O-O systems such as Stanford's Interviews library and some of Taligent's CommonPoint frameworks [15]. To reduce the amount of generated dynamic information, we rely on the coupling of static and dynamic information to perform selective class-wise instrumentation combined with the use of breakpoints to switch execution tracing on and off.

Human resources, on the other hand, are primarily related to the reduction of the cognitive load. For this, we also rely on the coupling of static and dynamic information to produce visualizations that combine dynamic properties with static properties and vice versa. Such filtered views allow the programmer to focus on certain aspects of system behavior while ignoring others. Finally, the GUI of Program Explorer relies on static-dynamic information coupling to produce hypertext-style integration between the different visualizations.

At the time of writing, three different prototypes of Program Explorer have been developed. In addition to the one described in this paper, we have developed a version based on debugging technology, which makes both program instrumentation and program database obsolete. The advantage of this system is a shorter edit-compile-explore turn-around time for the developers. However, the price paid is a greatly increased execution time. The third version, for IBM's System Object Model (SOM) [10], replaces program instrumentation with an extension to the SOM metaclass that captures method invocations [6] and uses the SOM repository for static information.

Currently, we are investigating the concept of **O-O Design Patterns** [7] with the goal of automating the processes of searching for and visualizing recurring designs in O-O systems. Viewing an O-O system from the perspective of design patterns often makes the detailed design more comprehensible. Our initial experience is that the static-dynamic coupling mechanisms described in this paper are very useful for pattern analysis [13]. The reason for this is that design patterns very often rely on "abstract behavior" defined in abstract classes but realized in concrete classes. We have found the vertical and horizontal selection technique described in Section 2 to be particularly useful for showing design patterns. If we can formally express the semantics of design patterns, we have the basic means for realizing automated search and visualization.

Even though Program Explorer is not intended for debugging, it demonstrates a clear potential for visual debugging. Its class- and object-centered visual representations of static and dynamic program information are an ideal communication medium for programmers. Moreover, its ability to keep a history of function invocations, as well as accesses to variables and the values of those variables, suggest the possibility of a more efficient debugging process, where programmers are able to investigate the events that lead to a run-time error instead of just being told where the error occurred.

Acknowledgements

We wish to thank our many colleagues at the IBM Tokyo Research Laboratory for their contributions to the Program Explorer project, and in particular Dr. T. Kamimura for his unflagging support. We are also grateful to R. Thornton and R. Pfeiffer of Taligent for their kind assistance in testing Program Explorer on Taligent's CommonPoint frameworks, and to M. McDonald of IBM Japan for checking the wording of this paper.

References

[1] H. Böcker and J. Herczeg. What Tracers Are Made of. In *OOPSLA '90, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* pages 89–99, 1990.

[2] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and Querying Software Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, 1992.

[3] M. Consens and A. Mendelzon. Hy$^+$: A Hygraph-based Query and Visualization System. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data,* SIGMOD Record, 22(2), pages 511–516, 1993.

[4] W. Cunningham and K. Beck. A Diagram for Object-Oriented Programs. In *OOPSLA '86, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* pages 361–367, 1986.

[5] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[6] I. R. Forman, S. Danforth, and H. Madduri. Composition of Before/After Metaclasses in SOM. In *OOPSLA '94, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* pages 427–439, 1994.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture.* Addison-Wesley, 1994.

[8] E. Gold and M. B. Rosson. Portia: An Instance-Centered Environment for Smalltalk. In *OOPSLA '91, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* pages 62–74, 1991.

[9] J. E. Grass. Object-Oriented Design Archaeology with CIA++. *Computing Systems,* 5(1), pages 5–67, 1992.

[10] IBM. *SOMObjects Developer ToolKit, Users Guide.* IBM Corp., 1993.

[11] M. F. Kleyn and P. C. Gingrich. GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views. In *OOPSLA '88, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* pages 191–205, 1988.

[12] C. Laffra and A. Malhotra. HotWire – A Visual Debugger for C++. In *Proceedings of USENIX C++ Technical Conference.* USENIX Association, pages 109–122.

[13] D. B. Lange and Y. Nakamura. Interactive Visualization of Design Patterns Can Help

in Framework Understanding. To appear in *OOPSLA '95, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* 1995.

[14] M. A. Linton, P. .R. Calder, J. A. Interrante, S. Tank, and J. M. Vlissides. *InterViews Reference Manual Version 3.1.* The Board of Trustees of the Leland Stanford Junior University, 1992.

[15] W. Myers. Taligent's CommonPoint: The Promise of Objects. *IEEE Computer,* 28(3), pages 78–83, 1995.

[16] G. M. Nielson, B. D. Shriver, and J. Rosenblum. *Visualization in Scientific Computing.* IEEE Computer Society Press, 1990.

[17] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the Behavior of Object-Oriented Systems. In *OOPSLA '93, Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* pages 326–337, 1993.

[18] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling Object-Oriented Program Execution. In *Proceedings of the 8th European Conference, ECOOP '94.* Lecture Notes in Computer Science, Vol. 821, pages 163–182, 1994.

A The Flyweight Sample Program

The program below starts with the creation of a session, a widget kit, and a layout kit. The widget kit is used to create a text label. A transformer is created and set to a 90 degrees rotation. In the session window, the label appears three times: normally, transformed (90 degrees rotation), and with a shadow background (see Figure 7).

```
int main()
{
   Session* session = new Session();
   WidgetKit& kit = *WidgetKit::instance();
   LayoutKit& layout = *LayoutKit::instance();
   Glyph* label = kit.label("HELLO");
   Transformer t; t.rotate(90.0);
   session->run_window(
      new ApplicationWindow(
         new Background(
            layout.hbox(
               label,
               new TransformSetter(label, t),
               new Shadow(label, 0, 0,
                          new Color(0.7, 0.7,
                                    0.7, 1.0)
                         )
            ),
            kit.background()
         )
      )
   );
}
```

Figure 7: *The Flyweight GUI.*