Generating Abstractions for Visualization

Steven P. Reiss and Manojit Sarkar

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-35
August 1992

Generating Abstractions for Visualization[1]

Steven P. Reiss and Manojit Sarkar
Department of Computer Science,
Brown University,
Providence, RI 02912, USA.
spr@cs.brown.edu, (401)-863-7641

Abstract

Modern technology allows us to view our programs through the visual representations that we normally use to describe and understand them. We are currently developing a new visualization system that will take advantage of today's technology to provide a flexible interface to a wide range of program visualizations. This system will allow the programmer to define visualizations as abstractions using queries over an object-oriented database of information about the program. These abstractions can then be visualized and browsed using easily defined type-based mappings and a generic filtering mechanism. This paper details the mechanisms and techniques we use to integrate the variety of information sources in a software system and to provide the programmer with a simple and convenient interface for defining abstractions.

1.0 Practical Program Visualization

Program visualization is the display of some aspect of a program. The display can be textual or graphical. The program aspect can vary widely: from source code, to a view of program execution, to a display illustrating the program's semantics, to

1. Support for this research was provided by the NSF under grants CCR9111507 and CCR9113226, by ARPA order 8225 and by ONR grant N00014-91-J-4052.

a design level view of the program. The program itself can range from a simple algorithm to a large, complex system.

Program visualization usually implies views that are automatically generated from either the program source or from its execution. Automatic generation enables visualization to provide the additional insights by showing the program from a different perspective without the programmer having to do all the work. The programmer already knows the program in terms of its source and its external execution behavior. Providing the programmer with conceptualizations that emphasize different aspects of the underlying system and that illustrate the internal behavior of the system provides the additional dimension that makes program visualization worthwhile.

The overall utility of program visualization has been limited. The major difficulty has been that the amount of information to be included is far more than can be displayed reasonably. The full call graph for the FIELD programming environment has over four thousand nodes and over twenty thousand arcs. A program dependence graph or a flow diagram for the system would be even larger. FIELD represents about a quarter million lines of source. A million line system, or a ten million line system, would be proportionally larger. Execution visualizations grow with the system's run time rather than with its size. Tracing all function calls or all the storage allocations or all memory access in even a simple program that runs for any substantial amount of time will yield megabytes or more of data to be analyzed and displayed.

Our goal is to make program visualization practical for real systems. Because it is impossible to view directly the large amount of information that is available, practical program visualization must provide tools to allow the user to select and display the information that is of interest. The user must be able to easily select the data to be displayed and indicate how this data is to be obtained from the underlying information.

A practical program visualization system can be achieved by focusing on abstractions. Abstractions are the basis for most of the ways that programmers look at their programs, data, and execution. They can be defined and generated along several dimensions: based on syntax, based on the underlying semantics, based on execution, or based on data structures. Examples of abstractions include the call graph and class hierarchy provided by Field, program slices, a State-transition view of a scanner, and a Petri net view of a distributed system.

Abstractions provides a reasonable intermediate form for program visualization. They allow us to represent the information relevant to the particular visualization while discarding the infromation that is not relevant. By using them as an intermediate representation, we can divide the problem of providing practical program visualizations into two parts. The first part is the definition of an appropriate visualization by the programmer. The second part is the generation of an appropriate visualization of an already defined abstraction. This paper describes

our methods for attacking the first problem. The task of generating appropriate visualizations from abstractions is covered in a separate paper.

2.0 A Framework for Generating Abstractions

Abstractions can be defined from a variety of sources of information about the program. Syntactic information such as the names of procedures, the locations of references and definitions, the call graph, or the class hierarchy, is generated either by compilers as with Sun's source browser or by a separate scanner as in Field or the Interviews class browser or CIA [5] or Genoa [4]. Symbol table information is stored as extra information in the executable file for debugging. Run time information such as the value of variables is obtained from the debugger while the program is executing or from a program trace afterwards. Information related to file dependencies, versions, and how the system was built is obtained from the UNIX tools *make* and *rcs*. Execution trace information is obtained dynamically while the program is executing using a monitoring facility such as provided in FIELD, or can be stored in a file for later monitoring as is done in AE [7] and the various UNIX profiling tools.

In order to allow the programmer to create an appropriate abstraction for later visualization, we need to provide a means of integrating this olio of information. The mechanism we use is a combination of control integration in the form of message passing as provided by FIELD [15,16] and data integration in the form of a program database. What distinguishes our approach from others is that our program database is a virtual entity and our use of a high-level object-oriented query language for defining abstractions.

The usual approach to providing a variety of program information to the programmer is to use a program database. By storing all information in a single database system, the underlying mechanisms such as the database query language can be used to automatically integrate information from a variety of sources. Such a database might contain, for example, parse trees generated by the compiler, semantic information such as use-def chains, configuration information, and version information. A single query can then be used to combine any of this information into a single abstraction. A program database is desirable because it offers a central store for all the data that can be used in an abstraction and provides a consistent interface to that data. Example of program databases include the OMEGA system [10], the cross referencer in FIELD, DIANA in Ada programming support environments [8], RPDE [6,9], PCTE [2], and abstract syntax trees in Reprise [17]. The varied and interconnected nature of the data that has to be stored in these databases have led researchers and implementers in the direction of object-oriented databases as the proper database model for this work.

While using a program database offers several advantages, it is not the desired solution. FIELD and other systems such as HP's Softbench, Sun's Tooltalk, DEC's Fuse, and SGI's Codevision are based on control integration. These systems have

demonstrated the power and convenience that can be achieved in an open environment based on existing tools. We would have to reimplement all our existing tools in order to integrate them tightly into a database-oriented environment. Moreover, such an environment would be dependent on a large, complex database system which would be subject to usage patterns far different from most database systems (updates, for example, would be more frequent than queries) and would make the environment overly complex. Existing tools have been designed to store and utilize the specialized types of data that they require. For example, the symbol table information required for debugging is stored in a compact form that is designed specifically to make debugging easier. It is relatively easy, using control integration mechanisms, to offer this specialized information from one tool to other tools in the environment.

An alternative to building an environment on top of a program database is to add a database system to an existing environment as a separate tool. Here the information that is generated by the existing tools would be stored, presumably by those tools, in a database system that would then serve as the basis for future queries. This approach has the disadvantage that the information to be stored in the database system is duplicate information. It is first stored in the current file-based form for use in existing tools, and second is stored in the database. Any scheme that involved replication of information introduces problems of consistency as well as those of generating and accumulating the information in the first place.

We wanted to maintain the openness of a control-based approach using existing tools while at the same time providing the necessary common access to the myriad forms of available data. Our strategy here is to allow the existing tools to do what they do best -- to act as specialized repositories for the information that is of interest to them. Then we build a virtual database that uses the message-based integration mechanisms of FIELD to access these specialized repositories. The virtual database consists of an object-oriented schema that includes all the information available from the various tools. It also includes a high-level object oriented query language over this schema. The query processor for this language maps the queries into appropriate requests for information from the underlying tools and integrates this information, storing it as a set of objects. This approach is similar to multi database systems [1].

The query processor takes information from the different tools and combines it to form objects that define the desired abstractions. The use of objects to define abstractions is based on our previous work with Garden [12,13]. Garden was an object-oriented programming system that provided a framework for conceptual programming. The underlying tenet was that programmers should be able to quickly define their own visual representations for specifying the solution to the problem at hand, and should be able to code in terms of these representations. Garden provided an object-oriented programming environment without an underlying language. Instead, the objects were directly executable and programmers could interactively define the visual presentation and the semantic behavior of the objects that characterized their representation. Garden was used to develop a variety of different

visual representations including state transition diagrams, data flow languages, tree transformation languages, Petri nets, port-based message passing, and data-based design languages.

One of the key concepts used in Garden was the use of the object-basis for language definition as the means for defining the visual syntax of that language. Garden introduced a set of tools that included a general purpose 2-D visualization package and two companion packages, one to allow the user to define the visual syntax for a given structure and one to allow the user to interactively edit the visual representation and map the edits into appropriate changes to the underlying structure [14]. These tools were used in Garden both for visualizing and editing the newly developed visual languages and for viewing and editing the data structures used by Garden programs.

The flexibility demonstrated by Garden for the definition of a myriad of different representations based on objects, and the demonstrated ability to take an object-based representation and provide easily defined mappings that can offer a quality visualization of the underlying objects directed us toward our use of a typed, object-based representation for abstractions. Using an object-based representation allows us to use an extension of the Garden mechanisms to provide high-quality visualizations of the abstractions we define. Our desire to use an object-based abstraction representation led us to adopt an object-oriented query language whose primary purpose was to define the structure of the objects that compose an abstraction and to specify how the contents of these objects can be derived from the information referred to by the object schema.

In the rest of this paper we detail how this approach is implemented as the initial basis for our abstraction visualization system. We begin by describing the object-oriented database schema that we have adopted. While this schema is currently limited to the information that is readily available within the FIELD environment, it is easily extensible to handle other sources of information. Secondly we describe the query language we have developed for allowing the programmer to define object-based abstractions using this schema.

3.0 A Consistent Object-Oriented Schema for Program Data

The first step in developing a consistent front end that integrates the local databases maintained by the different tools of a programming environment is to develop a data model. This model or schema must be powerful enough to handle all existing information sources. It must be extensible in order to handle new tools and hence new information sources in the future. At the same time, it must be simple and straightforward so that the programmer who has to define abstractions against it can understand it and use it effectively.

We are using an object-oriented representation to model this information. An object representation is powerful since it allows most data to be represented in a natural fashion. It is extensible since it can easily support new types of objects and

since it can use inheritance to allow the evolution and reuse of objects. It is also slightly more complex than simpler schemes, for example a relational view. However, we feel that we can make it simple enough to provide a programmer with an understandable and usable basis for defining abstractions.

In our schema, data is represented as typed objects. Each type is characterized by a set of data fields, a set of methods, and, optionally, a supertype. The data fields hold the information that characterize the type. For most of the basic types, two instances of the type with the same data field contents are considered the same object. Thus two files with the same (absolute) filename are considered by the database system as the same file. This is essential in attempting to integrate the data that is provided by multiple tools.

The methods associated with a type provide additional information about the object and are used for defining and evaluating queries. New methods for a given type can be added dynamically. Finally, our schema supports single inheritance. Inheritance is useful both for grouping sets of related objects (for example, the different components of the symbol table) and for allowing the schema to evolve as new tools are developed by introducing extended objects as subtypes of the original ones. We are not including multiple inheritance in our initial implementation, since we feel it has a limited applicability in this domain. However, we are considering it as an extension for future versions of the system.

3.1 Base Types

The model is built up from a limited set of base types. These include Integer, Boolean and Text. Integers and Booleans are provided with methods that support the basic arithmetic and relational operators. The type Text is used to hold arbitrary length character strings. It includes methods for different forms of string comparison including straight matching, case-insensitive matching, and regular expression matching. Two subtypes of Text are also included in the set of basic types. These are FileName and FunctionName. Filename supports methods for extracting the head, tail and extension of the file. FunctionName supports methods for extracting the class, method name and argument types for C++ method functions as well as for mangling and demangling names. To facilitate the description of objects, the object schema also provides for built-in and extended enumeration types.

Types in the object model can also be built up using the set constructor. The model includes the notion of a homogeneous set of objects. Methods on the set include membership testing and method mapping. The latter involves applying a single method to each element of the set and merging the results using a given initial value and a given operator. It can be used to check that there is a member of the set with a given property or that all members of the set have a given property. Sets can be treated as sets or lists, i.e. they can have an implicit ordering and non-unique elements or can be unordered with all duplicates eliminated. Set or list types are usually denoted by appending the word List to the type name. Thus the type

TextList denotes a list of Text objects. In the following, we assume that list types have been created for each of the basic types.

Type definitions are given in the form:

type_definition **::= TypeName** [supertype] **<** field { , field } **>**
| **TypeName (EnumName** { **, EnumName** } **)**
supertype ::= **SUBTYPE TypeName**
field ::= **FieldName : TypeName**

This form is used within this paper to describe the initial types. The notation with angle brackets identifies a standard type with the given data fields. The notation with parenthesis defines an enumerated type.

3.2 Cross Rreference Information

The initial types that are defined in the schema correspond to the information that is currently available in the FIELD environment. This information comes primarily from the cross-reference tool of the environment. This tool provides a relational database of information about the program that is obtained by scanning the source file (or, in the case of g++, by augmenting the compiler to output the necessary information). The tool provides, through the FIELD message server, a relational calculus query language to the stored information. Other information that is included in our initial model is available via query from the make/rcs interface and the debugger interface.

The basic types that the FIELD cross reference database describes are files, functions, declarations, references, scopes, calls, classes, and class members. The initial object schema provides objects for each of these. The object type

File<name:FileName>

represents a program source or include file. The methods that are provided for File objects include:

```
FileList includes();
FileList usedBy();
```

The information accessed by these methods is actually stored in the cross reference database as part of the file relation.

The object type

Function<name:FunctionName>

denotes a user defined function in the program. Methods that are provided for a function object include:

```
File file();
Integer line();
Scope scope();
```

```
Integer numArgs();
DeclarationList arguments();
Declaration functionDecl();
```

The first four of these functions return information that is stored in the cross reference database for each function. The function relation in the FIELD database also contains the list of argument names. Returning a list of the declarations corresponding to these names requires additional queries into the underlying database. We use methods to provide more convenient access to these commonly needed items in an object schema.

The object type

Scope<file:File,start:Integer,end:Integer,inside:Scope>

and its enumeration

ScopeClass(NONE,EXTERN,INTERN,ARGUMENTS,CLASS)

describe scope objects. Scope objects are provided by FIELD to facilitate finding the definition of an object corresponding to a given use since FIELD does not actually store this information but rather forces the programmer to compute it as needed. Our object schema provides methods that do this computation however. The methods provided for scope objects are:

```
ScopeClass scopeClass();
Boolean isInside(Scope);
Boolean accessible(Declaration);
```

The first of these returns the class of the given scope. The second computes the transitive closure of the scope nesting represented by the inside field. The third checks if the given declaration is accessible in the given scope, i.e. is both visible and not redefined.

Declaration objects are used in FIELD to describe the definition of the different types of program objects. In setting up the database schema, we had a choice of either forming a class hierarchy of such objects or of defining a single object class for all declarations. We chose the latter because we have found, in FIELD, that queries are often made that select one of a set of object types and such queries are better supported by having the type be an enumeration rather than inherent to the underlying structure. Thus the type definition for declarations is

Declaration<name:Text,scope:Scope>

Methods are defined for the the type Declaration return the information that the cross reference database saves for each declaration. These include:

```
File file();
Integer line();
Function function();
Type type();
DeclClass declClass();
```

where the enumerated type DeclClass is

DeclClass(NONE,STATIC,EXTERN,AUTO,REGISTER,TYPEDEF,
EXTDEF,PARAM,FIELD,EFUNCTION,SFUNCTION,
STRUCTID,UNIONID,ENUMID,CLASSID,CONST,MACRO,
FORWARD,VPARAM,FRIEND,PROGRAM,EXCEPTION,
USER_KEY,LABEL)

The object type

Reference<name:Text,File:file,line:Integer,function:Function>

denotes a reference to some object in the source program. The fields identify the object by name and give the location of the reference. The methods available for a reference object include:

```
Boolean isAssignment();
Declaration definition();
Scope referenceScope();
```

The first method denotes whether the reference is an assignment or not, information contained in the FIELD database. The second finds the declaration associated with the given reference using the Scope information provided by Field. The third finds the scope in which the reference occurs.

The object type

Call<from:Function,call:Text,file:File,line:Integer>

denotes a call site in the source. The target of the call is identified by its textual name because the actual target may not exist in the program and thus might not correspond to a Function object. (For example, a call of a system routine that was not otherwise declared.) The function object can be found, where it exists, using the built-in method:

```
Function callFunction();
```

This method involves finding the appropriate declaration of the given name that is a function object provided that object exists.

3.3 Configuration Information

The information provided by the make interface in field includes dependency information as well as the information gathered from rcs. The basic unit is the file contained in a directory or project. To facilitate the integration of this information with that provided by the cross reference tool, the two tools share the same basic object, File. The integration with this tool adds new methods to the File object. These include:

```
Directory directory();
FileBuildType itemType();
DependencyList dependencies();
Boolean isOutOfDate();
Boolean isChanged();
Boolean isPhony();
```

```
Boolean useSCC();
Text version();
Text lockType();
Text lockState();
Text lockOwner();
```

The enumeration type used here is

FileBuildType(NONE,PROJECT,COMMAND,SYSTEM,
INTERMEDIATE,SOURCE)

The type Directory is a subtype of File. It is a file in some sense, but, for the purposes of configuration management, has other associated information. The type is defined

Directory SUBTYPE File<>

and provides the methods:

```
File defaultTarget();
Text buildTool();
Text configurationTool();
File makefile();
```

The type Dependency is used to describe a dependency between files as seen by the configuration management tool. (The `usedBy` and `includes` methods provide the information from the point of view of the compiler.) This is defined as:

Dependency<from:File,to:File>

where the type Dependency has the following methods:

```
Boolean isExplicit();
Boolean isImplicit();
Boolean isRecursive();
```

that provide additional information about the dependency.

3.4 Debugging Information

The third tool that we currently integrate through the object schema is the debugger. The debugger reads the symbol tables produced by the compiler and has information about types. While the cross reference tool identifies types only by name, the debugger can provide detailed information about each type. To support this, we use a Type object. This is actually returned as part of the declaration query and thus supports the interaction of the tools.

The definitions of the type Type include:

Type<name:Text>

RecordField<fieldName:Text,fieldType:Type>

TypeClass(NONE,PRIMITIVE,FILE,SET,ARRAY,RECORD,UNION,
SAME,ENUM,POINTER,RECORD,RANGE)

We keep the type object itself simple so that it can be defined solely from the type name, and thus from information from either the debugger or from the cross referencer. To allow querying of the additional information that the debugger can provide, we introduce a set of method calls:

```
TypeClass typeClass();
Type baseType();
Type rangeType();
RecordFieldList fields();
TextList enumerants();
Integer rangeFrom();
Integer rangeTo()
```

Here the `baseType` method is used to provide additional information about file, set, array, same (typedef), and pointer types. The `rangeType` method denotes the range of an array while the `rangeFrom` and `rangeTo` methods allow the programmer access to the bound information for the array. The `fields` and `enumerants` methods provide additional details for record or union types and enumeration types respectively.

3.5 Performance Information

Finally, we include the performance front end in our set of integrated tool. This tool runs one of the profiling utilities, either prof, gprof, pixie on the Decstations, or our instruction count profiler iprof on the Sparcstations. It stores this information and provides access through the message server to it. The information that is obtainable from this tool is made accessible in the object schema through methods on File and Function objects. The methods for File objects include:

```
Integer timeUsed();
Integer numInstructions();
```

The methods that are added for Function objects include:

```
Integer numEntry();
Integer timeUsed();
Integer lineCount(Integer);
Integer lineTimeUsed(Integer);
```

Not all of these methods will return valid values in all cases, since they depend on the profiler that is actually used on a given system and on the way that the program was compiled.

4.0 An Object-Oriented Query Language for Abstractions

In order to compose interesting abstractions from the objects available in the database, we need a powerful language which is not only capable of retrieving information, but also has the capability of building complex new objects [3,18]. The object-oriented query language described in this section is designed to serve three functions. It can be used to compose abstraction objects by retrieving existing

information, to define new types, as well as to add additional methods to existing types.

4.1 Informal Syntax and Semantics

Abstractions are built using retrieval operations. The retrieval operation always builds a new collection of objects of a specified type. The form of this operation is as shown below:

retrieval_operation **::= OBJECT** object_basis [from_part] [where_part]
from_part **::= FROM** variable_defs
where_part **::= WHERE** boolean_expr

The syntax above is like the SQL SELECT-FROM-WHERE syntax. This SQL-like syntax is chosen because, SQL is widely used and the SQL syntax is very readable. The non-terminal *object_basis* in the **OBJECT** clause defines the type of the object resulting from the query, and optionally provides a **VariableName** for the new object. Subsequent queries can use this variable name to refer to the object.

The syntax rules for object_basis is:

object_basis **::=** [collection_qual] [**VariableName :**] type_spec
collection_qual **::= SET | LIST**
type_spec **::= TypeName** [supertype] [type_fields]
type_fields **::=** type_field { **,** type_field }
type_field **::=** [**VariableName =**] **FieldName** [**:** type]

Evaluation of a retrieval operation generates a collection of objects which have the type defined together by *collection_qual* and *type_spec*. The collection can be forced to be a set or a list by specifying appropriate collection qualifier. **TypeName** can name a pre-existing type, in which case field types can be ommitted. If the named type do not already exist in the system, the query creates it. **VariableNames** are the free variables eventually bound by the **FROM** and **WHERE** clauses of the query. If a VariableName corresponding to a FieldName is omitted, FieldName can be used as a VariableName for the field.

The **FROM** clause defines the domain objects for the query. It's syntax is:

variable_defs **::=** variable_def { **,** variable_def }
variable_def **::= VariableName IN** object_collection
| VariableName IS object
object_collection **::= TypeName**
| (retrieval_operation **)**
| CollectionName

The terminal **IN** causes the variable to iterate over all the objects in the object collection, and **IS** causes the variable to get bound to a single object. When a **TypeName** is specified as the object collection, the collection includes all the objects of that type and all its subtypes. When the object collection is the result of a retrieval operation or is a collection named by a **CollectionName**, the collection simply includes all the member objects. The collection named by CollectionName could be either a set or a list built by a previous retrieval operation.

The **WHERE** clause specifies a boolean expression involving the variables in the BUILD and FORM clauses. An omitted WHERE-clause is equivalent to WHERE TRUE. The form of the boolean expression is as follows:

```
boolean_expr    ::= boolean_expr OR boolean_expr
                  | boolean_expr AND boolean_expr
                  | NOT boolean_expr
                  | ( boolean_expr )
                  | object
                  | relational_expr
relational_expr ::= object RelationalOperator object
object          ::= ( retrieval_operation )
                  | postfix_expr
                  | LiteralConst
postfix_expr    ::= postfix_expr . FieldName
                  | postfix_expr . MethodName ( [ arg_list ] )
                  | VariableName
arg_list        ::= object { , object }
```

Available **RelationalOperators** are defined by the type of the left hand side of operand of the relational expression. Any character string for the builtin type Text, two boolean constants, TRUE, and FALSE, and any integer value can be used in place of **LiteralConst**.

New types can be added to the system by type definitions using the following syntax:

```
type_def    ::= TYPE TypeName [ supertype ] < field { , field } >
              | TYPE TypeName ( EnumName { , EnumName } )
              | TYPE TypeName Set < TypeName >
              | TYPE TypeName List < TypeName >
supertype   ::= SUBTYPE TypeName
```

field **::= FieldName** : **TypeName**

Type definitions add new types into the system. The syntax only allows definition of data fields at the time of type definition, methods can be added later. All the data fields and methods are directly accessible by the "**.**" operator. Two predefined type constructors Set and List provide abstract base types for set and list abstrcations. The types defined using the type constructor Set has the following predefined methods:

```
Boolean isMember(e: ElementType);
SetType diff(set: SetType);
SetType union(set: SetType);
SetType flatten();
Integer count();
```

Similarly all the types constructed using the List type constructor has the following set of predefined methods:

```
ElementType ith(i: Integer);
ListType sublist(i: Integer, j: Integer);
ListType catenate(list: ListType);
ListType flatten();
Integer count();
```

New methods can be defined for the objects using C++. The syntax is defined by the following set of rules:

method_def **::= METHOD TypeName ::** method_sig **{** method_body **}**

method_sig **::= MethodName (** **[** param_list **]** **) :** type

param_list **::=** param **{** **,** param **}**

param **::= ParamName :** type

method_body **::= TextConst**

The method gets attached to the type specified by **TypeName**. Method signature defines the return type, and the types for the method's parameters. The first parameter is implicit, and it is the object used to invoke the method. **TextConst** is a piece of C++ code which can refer to the data fields, and other methods associated with the invoking object, as well as those of the actual arguments. These methods are compiled and loaded into the database dynamically.

4.2 Example Queries

In this section we show a three example queries ranging from a very simple query to moderately complex queries. The following set of statements in the query language is written to build a simple call graph abstraction:

TYPE Arc<rerurn: Node>

```
TYPE Node<name: Text, calls: List<Arc>>
OBJECT SET simple_cg = Node<name, calls>
        FROM f IN Function
        WHERE n = f.name AND c = (
                OBJECT LIST Arc<return>
                        FROM c IN Call
                        WHERE c.from = f.name AND return = c.call )
```

The following query is a slightly more complicated example that builds a call graph for each file containing at least one function:

```
OBJECT SET cg_files = CGFiles<name: Text, nodes: Set<Node>>
        FROM file IN (OBJECT SET FuncFiles<file: File>
                                FROM function IN Function
                                WHERE file = function.file )
        WHERE name = file.name AND nodes = (
                OBJECT SET Node<name, calls>
                        FROM function IN Function
                        WHERE function.name = file.name AND calls = (
                                OBJECT LIST Callee<nm: Text>
                                        FROM cl IN Call
                                        WHERE cl.from = function.name AND
                                                cl.call = nm ) )
```

The following query returns a set of function names that modifies arguments having type “ControlPolicy”. The predefined method flatten() for the type constructor Set is used in this query to access objects within set of sets.

```
TYPE Args Set<Arg<name: Text, type: Text>>
OBJECT SET policy_assigners = Assigner<name: Text>
        FROM  ref IN Reference, arg IN (
                OBJECT SET args_set = Args<args: Set<Arg<name, type>>>
                FROM function IN Funtion
                WHERE args = function.arguments() ).flatten()
        WHERE name = function.name AND
                arg.name = ref.name AND
                arg.type = “ControlPolicy” AND
                ref.isAssingment()
```

5.0 The Query Processor

The query language defined in the previous section is implemented in the abstraction generator as a database front end. This involves parsing the query language, mapping it into database operations, optimizing those operations, and then executing the optimized query.

While parsing the query language is straightforward, translating it into database operations is not. The complexity in the translation process comes form the need to handle method calls efficiently, since the rest of the language can be translated using standard database techniques. Methods are used in the database schema for a variety of different purposes. Each of these requires them to be handled in a different way in terms of the translation. The different uses of methods are:

- *Access Methods*: Some of the methods do not necessitate any additional computation but merely access data that would be gathered from the cross-reference (or some other) database when any other information is gathered. For example, for Function objects, the `line`() method returns information that is part of each Function tuple.
- *Indirect Access Methods*: Other methods are similar to access methods except that the value they return is an object whereas the value stored in the database is a text string. Evaluating these methods implies finding or creating an internal object in the query processor that corresponds to the textual value that is otherwise available. An example of this type of method is the `file`() method associated with most of the types.
- *Query Methods*: Most of the methods that have been defined in the database schema involve making additional queries into an external database. Some of these are simple in that the they involve only a simple query that returns the actual information. An example of this would be the various performance methods such as `timeUsed`(). Others involve making the query and then creating or finding the appropriate internal objects such as the File method `includes`(). Others such as the `defintion`() method for Reference objects involve a complex database query.
- *Computational Query Methods*: Some of the methods require that the data obtained from a query be interpreted in various ways. For example, to compute the `isInside`() method for Scopes, zero or more queries might have to be done to handle transitive closure. Similarly, the `callFunction`() method for Call objects may involve several queries attempting to find the proper function for the calling scope given only the function name.
- *Computational Methods*: Some methods are simple in that they translate directly into internal database or arithmetic operations. Examples of these include all the methods on the built-in object types.

Each of these different types of methods must be handled differently. To accommodate them, each method is defined with its own translation procedure that takes the query expression and generates an appropriate database query tree.

The result of the translation process is a tree over a set of internal database operations. These include standard arithmetic, string and Boolean operations on expressions, as well as database operations such as Select, Product, Project, and Join that operate on streams of objects. Additional operations are defined to handle the more sophisticated method evaluations. The evaluation model is close to that used in a relational algebra based database system and builds on our previous experience with the Eris database system [11].

The operation trees are next optimized using multiple passes of a tree transformation based query optimizer. The optimizer uses estimated cost information that is associated with each operator. It also attempts to combine as much as possible all operations that are targeted to a single external system so that queries to that system can be combined if appropriate. The optimizer uses a dynamic programming approach to finding a minimal resultant operation tree. This is again based on our experiences with Eris.

The result of the optimization pass over the query tree is an executable database expression based on object streams, operations on object streams, and operations on individual objects. An interpreter takes this tree and does the actual evaluation. When information is needed from other tools, the requests are made through the FIELD message server. The details of this process of query optimization and evaluation are the subject of another paper.

6.0 Conclusions

Practical program visualization for large software systems requires that the user be able to define arbitrary abstractions of the system using all the information that is available. This information currently is scattered among the many tools that are used in the software system. Rather than gathering that information into a single database system and then having to rewrite all tools to use this system, we take a more open approach that integrates the existing tools using a common database interface.

The work reported here is only a first step toward generating complex abstractions. Additional work needs to be done in several areas:

- Some abstractions should be incremental in nature and the query definition and evaluation strategy should support them. For example, a dynamic call graph abstraction based on trace information generated from a running program is going to change as more information becomes available. The evaluation of the corresponding query should be done incrementally if at all possible.
- Additional and more sophisticated information sources need to be integrated into the schema. For example, we are currently working on providing source information based on annotated abstract syntax trees and on deriving semantic information in the form of program dependence graphs.

- Currently, each method is handled as a special case during the translation process. We should be able to more formally characterize the behavior of these methods so that the definition of a new method and its incorporation into the system can be more easily achieved.
- We need to obtain experience with the utility and power of the query language for its targeted application of defining abstractions. This should be more feasible later in the summer when then tools that provide visualizations of the generated abstractions are available.
- To support visual browsing over complex abstractions, the database system should support requerying, that is applying a secondary query to a previous result. This should be able to be done within the current framework, although many of the optimization criteria are different.

7.0 References.

1. Rafi Ahmed, Philipe DeSmedt, Weimin Du, WIlliam Kent, Mohammad A. Ketabchi, Witold A. Litwin, Abbas Rafii, and Ming-Chein Shan, "The Pegasus heterogeneous multi database system," *IEEE Computer* Vol. 24(12) pp. 19-27 (December 1991).

2. Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas, "An overview of PCTE and PCTE+," *SIGPLAN Notices* Vol. 24(2) pp. 248-257 (February 1989).

3. O. Deux and et al, "The O2 system," *CACM* Vol. 34(10) pp. 34-48 (October 1991).

4. Premkumar T. Devanbu, "GENOA - A customizable, language- and front-end independent code analyzer.," Artificial Intelligence Technical Report, AT&T Bell Labs (October 1991).

5. Judith E. Grass and Yih-Farn Chen, "The C++ information abstractor," *Proceedings of the Second USENIX C++ Conference,* pp. 265-275 (April 1990).

6. W. Harrison and H. Ossher, "RPDE3: An environment framework supporting change," IBM Thomas J. Watson Research Report RC17259 (Octovber 1991).

7. James R. Larus, "Abstract Execution: A technique for efficiently tracing programs," U. Wisc.-Madison Computer Sci. Dept. TR 912 (February 1990).

8. Robert Munck, Patricia Oberndorf, Erhard Ploedereder, and Richard Thall, "An overview of DOD_STD_1838A (proposed), The common APSE interface Set, Revision A," *SIGPLAN Notices* Vol. 24(2) pp. 235-247 (February 1989).

9. H. Ossher, "Multi-dimensional organization and browsing of object-oriented systems," *Proc IEEE Computer Society 1990 Intl. Conf. on Computer Languages* (March 1990).

10. Michael L. Powell and Mark A. Linton, "Visual abstraction in an interactive programming environment," *SIGPLAN Notices* Vol. 18(6) pp. 14-21 (June 1983).

11. Steven P. Reiss, "The efficient implementation of flexible relational database systems," Brown University CS-82-02 (1982).

12. Steven P. Reiss, "Working in the Garden environment for conceptual programming," *IEEE Software* Vol. 4(6) pp. 16-27 (November 1987).

13. Steven P. Reiss, "An object-oriented framework for graphical programming," pp. 189-218 in *Research directions in object-oriented programming*, ed. Peter Wegner,MIT Press (1987).

14. Steven P. Reiss, Scott Meyers, and Carolyn Duby, "Using GELO to visualize software systems," *Proc. UIST '89*, pp. 149-157 (November 1989).

15. Steven P. Reiss, "Interacting with the FIELD environment," *Software Practice and Experience* Vol. 20(S1) pp. 89-115 (June 1990).

16. Steven P. Reiss, "Connecting tools using message passing in the FIELD environment," *IEEE Software* Vol. 7(4) pp. 57-67 (July 1990).

17. David S. Rosenblum and Alexander L. Wolf, "Representing semantically analyzed C++ code with Reprise.," *Proceedings of the Third USENIX C++ Conference*, pp. 119-134 (April 1991).

18. Gail M. Shaw and Stanley B. Zdonik, "A query algebra for object-oriented databases," *Proceedings of the 6th Intl Conf on Data Engineering*, pp. 152-162 (1990).