MetaFlex:
A Flexible Metaclass Generator

Richard Johnson and Murugappan Palaniappan
Aldus Corporation
411 First Avenue South
Seattle, WA 98104-2871 USA
muru@aldus.com, richj@aldus.com

Abstract

Motivated to support the needs of component-based applications, we have developed a system called MetaFlex that generates metaclasses to extend the behavior of our C++ classes without inventing variants of the original classes. We make the case that a flexible metaclass generator service that allows developers to freely choose the kind and degree of detail for each metaclass is needed and present our architecture for making this specification. We also illustrate a powerful use of this technique with a scripting extension to our application framework. With an evaluation of our current MetaFlex implementation and its use with the scripting extension, we conclude that this service is best provided by compiler vendors.

Introduction

The 1980s was a period where many organizations developed application frameworks. We can expect the 1990s to be a period where development of component-based application frameworks will be prevalent. An industrial strength, platform independent application framework developed in C++ by Aldus was presented in [Ferrel89]. The next evolutionary step towards development of a C++ component-based application framework is presented in [Christiansen92]. This article describes one of the major issues in achieving the componentization goal - the need for accessing information about classes, usually lost during compilation, at run-time.

Component-based application frameworks allow users to construct an application dynamically to best meet their needs . By construct, we mean that users can install a set of components, at run-time, from a list of published components. A component can be as simple as a single drawing tool or as complicated as a page-layout application. The application framework provides the necessary glue for components to work together as an application. The glue is developed without knowing what kinds of components would be developed and published.

As part of the componentization goals, we desire the ability to dynamically extend an application. An extension, in our sense, is the ability to add behavior to a class either statically or dynamically without structurally modifying the nature of the original class

being extended. Examples of extensions include:

- **Scripting.** Usually there is a delay between application functionality development and the development of end-user scripting access to this functionality due to the orthogonal nature of scripting development and application functionality development. By using parser technology for generating an application's metaclass hierarchy, it is easy to expose the application functionality during its development for end-user scripting.

- **Database access control.** In terms of the access protocol to a database, the primary difference between single-user databases and multi-user databases is the need for transaction serialization and user access privilege management. Database access control semantics are orthogonal to this base functionality and are appropriately added as extensions.

- **Test monitors.** Unobtrusive introduction of "up-stream" quality assurance methods may be one of the most important usages of our proposed extension architecture. There are countless applications, including test coverage tools, memory and performance profiling, semantic assertion checking, message flow monitors for dynamic program behavior analysis, and object inspectors.

We have implemented a scripting extension as part of our component-based application framework development and will use it for the case study presented in this paper. We wanted to build a scripting scaffold into the application framework such that the code can manipulate object types that are not known at compile-time but can be determined at run-time. Object types refer to information like the addresses of the objects' class methods and member variable map information. While not universally true, many typed language implementations, including most commercial versions of C++, do not preserve the class type information. By discarding it, they effectively freeze, at compile time, the data maps and functions that operate on it.

To overcome this limitation, we developed a tool architecture called MetaFlex, and have a production version of it by the same name. MetaFlex is a *metaclass* generator that is built on top of a C++ parser. Metaclass refers to the type information of a class extracted at compile time and made available at run-time.

The focus of this paper is to describe the MetaFlex architecture and the experiences we had in achieving the scripting extension goal of the application framework using MetaFlex. Two additional capabilities, though not implemented in the production version of our current MetaFlex tool, are described: 1) a means of specifying what type information should be made available in the application on a class by class basis, and 2) a mechanism for dynamically attaching code and data to extend class behavior.

MetaFlex is more general than the metaclass notion in Smalltalk [Goldberg83] and the factory notion in Objective C [Cox86]. MetaFlex allows extension developers to specify

exactly what type information is needed and how much is needed, properly gauged to the application being developed whereas Smalltalk and Objective C automatically generate metaclass (or factory) information for all classes of the application being developed. We argue that a MetaFlex-like tool is needed for creating the types of extensions we are suggesting because a) the metaclass requirements for one extension may be quite different from that of another, b) we agree with [Stroustrup92], that it is not possible to generate an "ideal" metaclass, and c) the generated metaclass information should be as small as possible to reduce the size of the final executable. This final point, while perhaps less an issue on sixty-four megabyte workstations, is definitely a problem when the target environment is a PC class machine.

With this context in mind, the goals of this paper are to:

- Make the case that a flexible metaclass generator service is needed that will allow developers to freely choose the kind and degree of detail for each metaclass to be made part of an application.

- Explain how the use of a generated metaclass offers the developer a means to dynamically add orthogonal extensions of class behavior with mixin classes, minimizing the need to develop specialized versions of the class.

- Describe the architecture we have used to design and implement a flexible metaclass generator service, highlighting issues that may be useful for other developers of metaclass generator services. An architectural description is included of a proposed metaclass specification grammar.

- Illustrate a powerful use of this technique with a scripting extension to an application framework.

We first present related work in the area of metaclass generation services and extension application examples. Then, we present our design of the MetaFlex architecture which includes a description of our specification language. We next describe the scripting application we use as a case study of an extension application, followed by a description of the MetaFlex implementation for the scripting application. We conclude with an evaluation of our implementation, highlighting issues that may be useful for developers of metaclass generation services.

Related Work

Cointe's notes that metaclasses provide meta-tools to build open-ended architecture [Cointe87]:

> "From an implementor's point of view, metaclasses are very powerful because they provide hooks to extend or modify an existing kernel. Metaclass allows the programmer to provide a flexible means to introduce inherited behavior in a system, including single and multiple inheritance as well as method wrapping."

Since our applications are primarily developed in C++, we can also choose whether these extensions are to be dynamically bound to the application at execution time or are generated, compiled and made a static part of the system. This flexibility is possible because of the nature of our metaclass design.

The creators of C++, in the process of developing their own run-time type system implementation, acknowledge the predicament of creating a metaclass that would meet all engineering needs [Stroustrup92]:

> "... the likelihood that someone can come up with a set of (run-time type) information that satisfies all users is zero".

For the most common computing platforms found in use today, it is too expensive in terms of time and space to make all possible type information about a class available at run-time. More importantly, the requirement for *what* type information is needed by a given application is highly dependent upon the application being built.

Although primarily focused on database extensions, a similar architecture is suggested for extending class behavior [Wells92a]:

- Use a meta-architecture consisting of a collection of glue modules and definitions to provide the infrastructure for specifying/implementing event extensions and regularizing interfaces between modules.

- Develop an extensible collection of extender modules that implement OODB functionality via behavioral extensions.

Our metaclass architecture roughly corresponds to the authors' notion of glue modules while our Extension classes roughly corresponds to their notion of extender modules. The authors see a robust implementation of their Open OODB architecture having extenders for transactions and access policy management, distributed access to objects, object versioning, and database index maintenance, among others. This is on target from our perspective, but stops short in that it constrains the domain of usage of this mechanism to database functionality. Their initial implementation of the glue is the notion of a wrapper class that will assume the role of the class being modeled in the system. These role player classes are derived from base class Sentry, which duplicates the public interface of the modeled class as well as having metaclass protocol for installing, de-installing, and manipulating the extensions.

In discussing her work with the 3KRS language environment, Maes observes that the "extra structure and computation necessary to provide objects with special features such as documentation, constraints, or attachment (of other behavior) do not have to be supported for all objects in the system, but can be provided on a local basis" [Maes87].

This is a key architectural goal of MetaFlex. Why generate metaclass containers for types that are not to be extended?

Maes also explains that either an object can call upon its meta-object to extend or modify its overall behavior (conscious reflection), or presumably, other knowledgeable system objects can call upon an object's meta-object to modify the behavior of the object it models.

Our architecture differs from the 3KRS language environment in that we generate only one metaclass for each class. Although we have organized our metaclasses as first class objects, we don't create one for each class instance created.

One of the thrusts in our business of software development is to somehow reduce the tremendous expense associated with testing our applications before they are sold. In their work, Böcker and Herczeg introduce un-obtrusive tracing and profiling facilities to monitor the application developer's code [Böcker90]. The authors point out it is hard to make the tracing tools found in the marketplace "provide just the right amount of information at a level of detail that is just about right for the problem at hand." They recommend direct control over the specification of what to view and what to trace. At the same time, they recommend the extended methods be hidden from the programmer. In our architecture, these test monitors would be mixed in as extensions in the metaclass.

Kleyn describes a tracing tool that allows developers a means of visualizing the message passing between objects in their system [Kleyn88]. This application would make use not only of the metaclass extension architecture and the type information of the class being modeled, but also use the class hierarchy / lattice information collected by it or some other parser based application.

Palay describes ΔC++ [Palay92], a C++ system that permits the developer to release new compatible versions of libraries or dynamically loaded components without recompiling portions of the system that make use of the classes defined in these new components. The stated purpose of the work is to develop an environment that supports the full C++ language specification as well as dynamic loading of new functionality without performance degradation.

Achieving flexibility without performance penalty is one of our key requirements as well. Palay points out the ability to extend behavior dynamically allows one to quickly respond to changing application requirements, and that these extensions can be codified in a more optimal implementation later.

Murray describes Alf [Murray92], a system that represents C++ programs as trees of abstract objects. These abstract objects encapsulate the syntax tree complexities (offering a surrounding infrastructure) tailored for access by tools that use the trees as their input. Tools may attach tool-specific attributes to the Alf abstract objects, without affecting other tools that may have interest in the same abstract objects. Alf maintains these trees

permanently. The conventional file based organization of C++ source code is replaced. Incremental compilation and editing is implemented without compromising the static semantic analysis properties found in a conventional C++ compiler.

Although MetaFlex doe not work off a database, the notion of marking classes of interest with an Alf-like attribute is functionally equivalent to the metaclass specification we are suggesting in this paper.

In the next section, we first present the motivation for developing MetaFlex. Following it, we describe our parser architecture that constructs syntax trees of C++ source code. We then describe the MetaFlex architecture that generates metaclass information for a specific extension, based on the syntax trees it provides. The architecture section concludes with a description of our metaclass specification language.

Architecture

Motivation

Consider the bottom half of Diagram 1, below the dotted line. It depicts how a linker would statically compose a component based application. Nothing precludes dynamic system composition, however. If one were to replace the linker with a loader component that was part of a component framework kernel, it is not difficult to see how a component based system could be dynamically composed at execution time. The description of how the loader component might work is beyond the scope of this paper.

In our architecture, an extendible application, whether created statically or dynamically, is made with the following ingredients:

- metaclass library of the extendible classes (compiled metaclasses of the application)
- metaclass run-time library (various styles of function dictionaries, field dictionaries, component loaders, etc.)
- application libraries (statically or dynamically installable components, framework libraries)
- extension code (behavioral extensions to a class)

In both the static and dynamic cases, the key ingredient that supplies the necessary flexibility and extensibility properties is the run-time type information, or metaclasses for the application. The source of the metaclasses is depicted in the top half of Diagram 1, the MetaFlex generator, and shall be discussed in the next several sections.

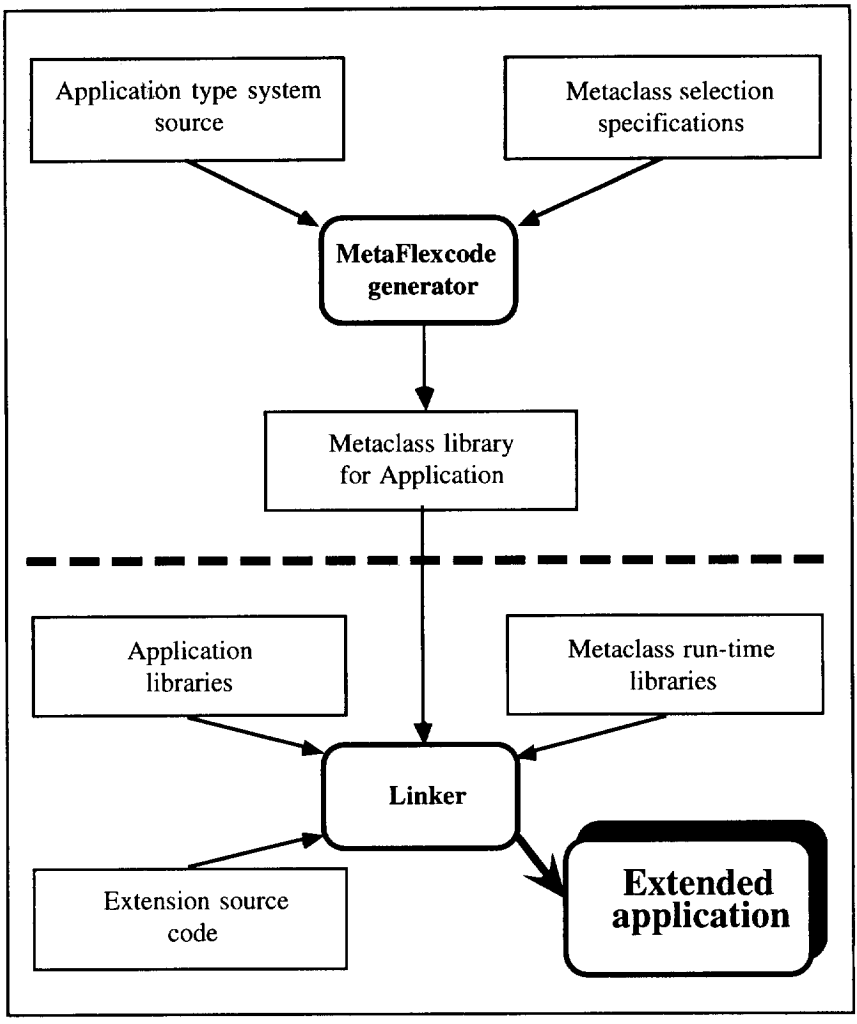

Diagram1 - Basic flow of MetaFlex metaclass information collection for statically linked component based application.

The MetaFlex Parser

In order to generally analyze and interpret C++ source code for purposes other than generating machine instructions (e.g., MetaFlex), one needs a C++ parser. In 1990, the availability of an inexpensive, commercial C++ parser for these purposes was non-existent. Those available were not able to deal with the full language syntax. In particular, we needed a parser that was capable of dealing with old-style K&R grammars through C++ 2.x grammars [Ellis90].

Since C++ is not an LALR(1) grammar, YACC can not be easily used to generate a parser

directly for C++. The solution that we selected was to use YACC to generate a parser for a modified version of its own grammar definition language. The generated parser will then read the C++ language expressed with this grammar, generating the parse tables that represent the follow symbol sets that guide the match process.

The parser we have built can be characterized as top-down, *for the most part predictive*, recursive descent. The *for the most part predictive* description is needed, since the parser does back-track in several situations. To minimize back-tracking during the match process, a dependency mechanism is employed. Dependent functions gain control at propitious moments. With the ability to freely access information in the token stream, symbol tables, and current syntax tree state, these functions force pre-mature match failures, whenever it can be determined that pursuing some sub-tree would be pointless.

Rather than mangle type names in the system and maintain a flat symbol table, we chose to implement a scoped symbol table, implemented as a stack of symbol tables built for the types previously seen in the input stream. These tables are pushed and popped from the scoped symbol table as needed.

The parser engine automatically traverses matched syntax trees for each compilation unit, giving control at the appropriate moments to the attached applications. The architecture of the parser permits multiple applications to be hosted or attached to it. Attachment consists of a parser application showing interest in a particular syntax tree node by adding an instance of itself to one of the node's dependency lists. Dependency list support is available on all node types, including rules, productions, and production symbols. It is possible to attach applications *before* a node or *after* it. This means that as the parser traverses a parse tree for its attached applications, these applications can receive control just before or just after a given sub-tree has been traversed. Furthermore, each application can show interest in more than one syntax tree node. This makes it convenient to collect information in complicated trees.

For certain C++ productions, parser information collection applications are implemented as part of the standard parser. In particular, the class declaration receives this treatment. For each class, an ordered collection of class member information is organized. Standard information, such as member name and base type of the member are always extracted, available through simple access protocol in the class information collector. If necessary, however, an application is able to access the syntax trees directly through the ordered collection member.

MetaFlex Source Code Generation

Built upon the parser described, we designed and implemented a flexible metaclass generator called MetaFlex to support the extension behavior of C++ classes. It uses the class declaration of a marked class to automatically generate code for a companion metaclass class. This metaclass class models the type information of its counterpart, the selected class. Although not part of our current implementation, our architecture plans

for specification of extension installation as well. These extensions may be used to either replace or extend the modeled class' behavior. An application's metaclass organization is similar to the Smalltalk approach, except that not all classes necessarily have metaclasses prepared for them. The application developers must explicitly choose which of the classes or class hierarchies need to be modeled with metaclasses.

Perhaps the key distinction to be made here is that the developer may not only specify which classes shall have metaclasses, but also choose which type information properties are to be generated for a given application class. For those properties selected for inclusion in the generated metaclass, it is possible to plug in different type information property representations for them. Different representations are desirable to avoid carrying large amounts of detail, when it is not needed or used in the application. For example, class type properties such as its method addresses and instance variable map information are available in several styles, selectable by the developer.

The following list summarizes the sorts of behavior for which we feel the metaclass should be responsible:

- the ability to create an object by name
- offer dynamic method dispatching by name to methods of a given class.
- the ability to extend and even change the behavior of one or more methods found in a given class.
- the ability to broadcast messages to all instances of a given class.
- offer tailorable class structural information to support service extensions, such as general purpose object streaming, persistent object management functionality, and scripting.

The metaclass instance creation protocol can be used to create instances of the type being modeled. If specified, the instances created by this protocol can be registered in an instance collection. No provisions in the architecture are made for registering instances of the objects created in the traditional manner, via the *new* operator. To make use of the registration service, the client must use the object creation service provided in the metaclass. Like other type information, instance registration service may be included on a case by case basis.

Code is generated by MetaFlex that automatically registers generated metaclasses in an application's metaclass dictionary, each entry keyed by the class name it models. This makes it possible to polymorphically create metaclass modeled objects (i.e. create these objects by class name), potentially by objects or processes external to the application. Polymorphic instance creation is part of the minimal, or what we refer to as the *vanilla* MetaFlex metaclass behavior.

In addition to flexible specification of the type system information to be included in the metaclass, a dependency list mechanism is provided by MetaFlex, similar in functionality to the Object dependency list behavior in Smalltalk. It is used to add E*xtension class*

derivatives to the Metaclass instance modeling a given class. As with the other type information, the presence of this feature is specified with instructions to the MetaFlex metaclass generator.

Extension class derivatives allow the developer to implement orthogonal behavior that extend the MetaFlex modeled class in some way. Since this extension is added to the class' metaclass, and not directly to the class itself, the original semantics of the class remain pure to their original purpose. Much as the Smalltalk View subclasses explicitly know everything about the Model subclasses they depend upon, the Extension class derivative has explicit access not only to the class instance being modeled, but also explicit access to the type information available in the metaclass as well. The reverse is not necessarily true. Nothing precludes an extended class from explicitly knowing about its extensions, but typically an omniscient configuration/policy object is given the responsibility to activate and deactivate the extended behavior.

Rather than create derived classes from a base Metaclass class to add extended type information about classes, we have chosen to develop a MetaInfo hierarchy instead. Each Metaclass *has a* MetaInfo. The MetaInfo contains references to the extended type information and developer supplied extensions required by the application. Diagram 2 illustrates the general relationships of the class, its Metaclass, its MetaInfo, extended type information, and developer-supplied extensions. The Class has a reference to its Metaclass. In our system, this is implemented as a class variable (i.e., a static variable in class scope). A reference to the metaclass of the modeled class' superclass(es) is also maintained in each metaclass. The MetaInfo instance may contain references to class extensions, class function information, class field information, and class instances. In order to be generated, the Metaclass, its MetaInfo, and the aspects of the MetaInfo must be specified to the MetaFlex code generator for the class in question.

Although it is not shown for each instance of the Extension classes in the MetaInfo's Extensions collection, there is a reference to the MetaInfo (and therefore Metaclass) for each Extension instance. We see most extensions being heavy users of a class' type information to perform its tasks. Applications like those suggested in [Richardson92], [Wells92b], [Pérez92], [Böcker90], [Voss92], and [Voss93] could be implemented as kinds of class Extension. These applications all use the type information of the class to which they are attached.

We have described the internal MetaFlex architecture in this section. In the next section, the external interface of MetaFlex is explained. This is the interface that extension developers will use to specify what types of meta-information they need.

Metaclass Specification Language

The following description of the Metaclass Specification Language is a suggested design and, at the time of this writing, has not been implemented in our existing MetaFlex tool. These notions, however, get at the details of what we think is needed.

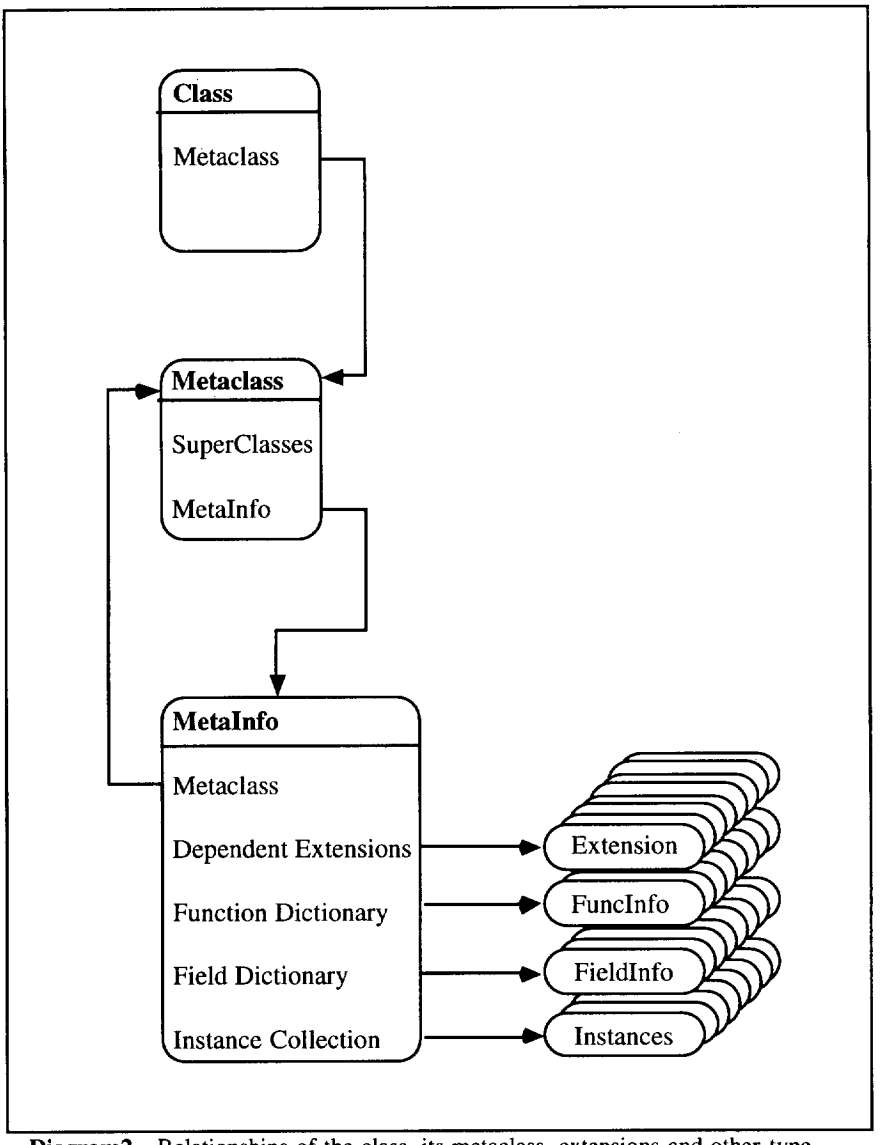

Diagram2 - Relationships of the class, its metaclass, extensions and other type information.

The specification of metaclass requirements to MetaFlex has been one of the more difficult areas for us. These requirements are ideally directly associated with the class declaration to be modeled with a metaclass. It is also desirable that these metaclass specifications do not interfere with the readability of the class declaration and its semantics. The reasoning here is that although a class can have explicit knowledge of its metaclass, as is suggested in Diagram 2, the knowledge is not necessary to understand the core abstractions the class provides its users.

Clearly, development environment support for viewing class declarations with and without the metaclass specifications is desirable. We also feel it would be useful to view all metaclass specifications together as a whole for an application. Such support does not exist commercially for C++. User definable attributes on ALF trees, as discussed in [Murray92], coupled with multiple view support on the class trees to which they are attached, would provide the sort of facility needed.

To control the composition of Metaclass derivatives a MetaFlex tool creates for us, we have developed a small metaclass specification language expressed in a YACC-like grammar as shown in Diagram 3. The grammar is admittedly elementary, using a simple keyword-value approach. It doesn't capture all of the possibilities by any means, but offers the reader a structured means of thinking about metaclass specification.

Given this language, the following four examples are representative metaclass specifications:

```
1. MakeMetaClassFor     Class       DatabaseObject
                        Support     FunctionDictionary Style
                                    FunctionAddressesOnly
                        Install     DBAccessPolicyManager

2. MakeMetaClassFor     Class       ApplicationKernel
                        Support     InstanceCollection
                        Install     ScriptingExtension

3. MakeMetaClassFor     AllDerivedClassesOf  Command
                        Support     FunctionDictionary Style
                                    FunctionDispatching
                        AllowAccessTo  PublicMembers

4. MakeMetaClassFor     Class       EventHandler
```

In the first example, a class named DatabaseObject will have Metaclass and MetaInfo class derivatives prepared for it. The MetaInfo class is created, since support for a function dictionary and a behavioral extension have been requested. The particular implementation of the FuncDict to be generated contains the address of the functions in the class for each member, but no extensive argument and return type information needed by applications such as scripting.

An instance of the Extension class derivative called DBAccessPolicyManager is to be added to the Extensions list in the MetaInfo object. Assuming an access policy manager design similar to that discussed in [Richardson92], the function dictionary with function address availability for each dictionary member would be needed to create the necessary method wrappers for the DatabaseObject member function protocol.

```
translationUnit          :  metaClassSpecifications
                         ;
metaClassSpecifications  :  metaClassSpecification metaClassSpecifications
                         |  metaClassSpecification
                         ;
metaClassSpecification   :  MakeMetaClassFor classParm specificationParms
                         ;
specificationParms       :  specificationParm specificationParms
                         |  specificationParm
                         ;
specificationParm        :  Support supportValues styleParms_opt accessParms_opt
                         |  Install extensionClassName
                         ;
classParm                :  Class CLASS_NAME
                         |  AllDerivedClassesOf CLASS_NAME
                         ;
styleParms               :  Style styles
                         ;
styles                   :  functionDictionaryStyles
                         |  fieldDictionaryStyles
                         ;
accessParms              :  AllowAccessTo accessControl
                         ;
accessControl            :  PublicMembers
                         |  ProtectedMembers
                         |  AllMembers
                         ;
extensionClassName       :  CLASS_NAME
                         ;
supportValues            :  NoMetaInfo
                         |  FunctionDictionary
                         |  FieldDictionary
                         |  InstanceCollection
                         ;
functionDictionaryStyles :  RespondsToSupportOnly
                         |  FunctionAddressesOnly
                         |  FunctionDispatching
                         ;
fieldDictionaryStyles    :  OffsetsAndLengthsOnly
                         |  FullFieldInformation
                         ;
```

Diagram 3 - Metaclass style specification grammar. Note that all grammar rules start in lower case. Grammar tokens all start in upper case. The _opt suffix on the end of several symbols is meant to suggest that occurrences of this symbol are optional.

In the second example, a class named ApplicationKernel will have Metaclass and MetaInfo class derivatives prepared for it. The MetaInfo is needed to support an InstanceCollection and ScriptingExtension extension instance.

The third example shows how metaclass requirements might be set for all classes derived from a given class. In this case, all derivatives of class Command will have a Metaclass and MetaInfo built for them. The MetaInfo will have a function dictionary capable of supporting function dispatching. Notice that more type information is required to support function dispatching than offered by the other function dictionary styles. Only the public members of the Command class derivatives will have their functions included in this dictionary.

The fourth example illustrates the case where vanilla Metaclass services (e.g., instance creation by name, instance type identification, instance length) are needed for a class named EventHandler, but no additional MetaInfo is required. Consequently, MetaInfo will not be generated for this class by MetaFlex.

In all examples, the application using the metaclass system, and in particular, the type of extensions attached to a given class' meta information, highly influence what sorts of type information should be made available at run-time.

Nothing in this architectural description precludes the use of dynamic loading of the meta information or developer-supplied extensions at run-time. In particular, the Extension, FuncDict, and FieldDict derived class instances could be streamed in from persistent storage. We see this architecture unifying the platform specific techniques used today to load class dynamic behavior.

In the next section, a description of the scripting application that uses the MetaFlex tool is presented, followed by a description of the MetaFlex tool currently in use.

Case Study: Scripting Enabled Applications

As part of the development of our component-based application framework, we wanted to provide scripting support for applications at the framework-level. An application-framework class, called the ScriptManager, provides the necessary support for processing end-user scripts to access, create, and change the contents of applications. If additional scripting functionality is required by an application that is not defined in the framework, it can bring along its own custom scripting.

In our application framework, we established ground rules for applications where all creation or content change operations are done through commands whereas content accesses of an object are done through directly though their method invocation.

At the time of writing this paper, we have implemented support for Apple Events [Apple91]. End users can use scripting applications such as AppleScript and Frontier

(they translate scripts to Apple Events) to access and manipulate applications. The three major Apple Events supported by the Script Manager, but developed without knowing the scripting requirements of the applications are:

- **Set event** - changes content of applications. For example, "set object foo to move:{10, 15}" is an instruction to the Script Manager to access object foo and dispatch a message to move it by the specified amount. The Script Manager translates this instruction to create a MoveCmd(foo, Point(10, 15)) and invoking its DoIt() method. In the DoIt() method, the command sends the message, foo-> Move(Point(10,15).
- **Get event** - accesses content of applications. For example, "get fontSize of object 4" is an instruction to access object 4 for its fontSize. This instruction translates to dispatching the message return(object 4->GetFontSize()).
- **Create event** - create contents in applications. For example, "create new line with color: green" is an instruction to create an object of type *line* and set its color to green. This instruction translates to creating a CreateCmd(newLine), invoking its DoIt() method. In the DoIt() method, the command created a newLine object and returns it to the Script Manager. Once the newLine is created, the Script Manager creates a ColorCmd(line, green) and invokes its DoIt() method. In the DoIt() method, the command sends the message, newLine->DoIt(green).

In the examples above, the complication is once the target object is accessed, the Script Manager needs a mechanism to determine whether the object will "understand" the message that will be sent to it. In the first example, whether object foo will understand the message Move() and in the second example, whether object 4 will understand the message GetFontSize().

In order to accomplish the goal of type-independent script dispatching by the Script Manager, we chose to use the MetaFlex tool to generate metaclasses for these objects. From the generated metaclass information made available at run-time by MetaFlex, the Script Manager uses the type information about an object to choose the correct command to change/create content or to access content of an object through its method invocation.

The following code fragments illustrate the interface between the Script Manager and the metaclass, generated by MetaFlex. What is shown in the code segment is only the interface between the Script Manager and MetaFlex and not the error checking or other processing that happens in the code. The first segment shows the Set event handler, next the Get event handler, and finally the Create event handler.

```
• Set event - changes content of applications.
{
        // Non-metaclass related code

        // Do the command dispatching by "Make"ing the command through its
```

```
        // metaclass.  Since it is a command, call its DoIt() method....
        // Note: NumArgs + 1 is passed in to specify the number of data values
        // plus the object descriptor
        Metaclass* aMeta = gTheApp->GetMetaclass(cmdName);
        MetaInfo* anInfo = (aMeta ) ?
                        (aMeta ->GetMetaInfo()) : (MetaInfo*)NULL;
        if(anInfo) {
                aCmd = (UndoableCmd*) anInfo->Make(numArgs+1,argv);
                if(aCmd) {
                        aCmd->DoIt();
                }
        }
        // Non-metaclass related code
}
```

• **Get event** - accesses content of applications.

```
{
        // Non-metaclass related code here ...
        // anObj is the object from which we want to access the information
        // cmdName is the method name to invoke in the object
        Metaclass* aMeta =anObj->GetMetaclass();
        MetaInfo* anInfo = (aMeta ) ?
                        (aMeta ->GetMetaInfo()) : (MetaInfo*)NULL;
        FuncDict* aDict = (anInfo ) ?
                         (anInfo ->GetFunctionDictionary()) :
                        (FuncDict*)NULL;
        if (aDict) {
                void* buffer = (void*)
                        (aDict->Execute(anObj,cmdName,numArgs,argv));
        }
        // Non-metaclass related code
}
```

• **Create event** - create contents in applications.

```
{
        // Non-metaclass related code here ...
        // Get the command that will create one of this object
        Metaclass* aMeta = (Metaclass*)(gTheApp->GetMetaclass
                                                (theCmdName));
        MetaInfo* anInfo = (aFact) ?
                         (aMeta ->GetMetaInfo()) : (MetaInfo*)NULL;
        if(anInfo) {
                aCmd = (UndoableCmd*) anInfo->Make(numArgs+1,argv);
                if(aCmd) {
                        aCmd->DoIt();
                }
        }
        // Non-metaclass related code
}
```

The interface requirements specified by the scripting application to the MetaFlex are:

- generate metaclasses for specified commands and objects
- provide ways of querying which methods are understood by a given class
- determine return type, function name, and argument types for all functions found in classes modeled by metaclasses.
- provide an interface to create commands and initialize them
- provide an interface to execute a method of a selected object and return its value

As we had not implemented the metaclass specification language described in the architecture section, we used a macro DEFINE_METACLASS to make the specification to MetaFlex. This macro introduced the static variable gMetaclass into the declaration, and is the key used by MetaFlex to decide whether or not to generate a Metaclass derivative for the class. From the application developers' perspective, all they need to do is mark which commands and objects they want to give access to end-users, with the macro.

MetaFlex Tool Implementation for the Scripting Application

Compile-time Description

Metaclass class declarations for the specified (i.e., marked with the DEFINE_METACLASS macro) classes of the scripting application are generated by the current version of MetaFlex. MetaFlex generates several source code files containing the metaclasses. Depending upon the number of classes specified to have metaclasses generated for them, more than one source file may be generated to hold the metaclass declarations, definitions, and instantiation code. This feature was added due to compiler capacity limitations. These source files are then compiled with the regular C++ compiler. The compiled object code is linked with the application's object code created from the original source, to form the extended application. The generated source for the scripting application is described in the remainder of this section.

Since function dispatching by name is the key functionality needed by the scripting application, MetaFlex generates MetaInfo class declarations containing references to function dictionaries. The type of function dictionary generated is able to dispatch the functions of the class it is modeling. Field dictionaries, instance collections, or the Extension class described in the architectural section are not used. Function dispatching is done on behalf of the Script Manager by explicitly accessing the object's function dictionary found in its MetaInfo instance.

Beside generating class declarations and their implementations, MetaFlex also generates instantiation code, that can create instances of the Metaclass, MetaInfo and FuncDict classes. Of particular note here is a C function generated for each FuncDict initialization. Initialization of the function dictionary is done at its first access to minimize the application startup time. The initialization process consists of loading the FuncInfo

elements with the function member information of the class, including the function address. For example, given the member function declaration of class foo,

```
void            Init (type1 var1, type2 var2);
```

the following pointer to member function variable [Ellis90], named ptrMem, is generated by MetaFlex:

```
void (foo::*ptrMem)(type1,type2)    =    &foo :: Init;
```

To support function dispatching, MetaFlex must do a reasonably good job understanding the types used in the member function declarations that are being included in the function dictionary. Not only the string representations of a member function's base type and argument types must be known, but the base type of all typedefs used must be understood as well. The base type understanding is needed to determine argument lengths and whether they are references.

Run-time Description

The run-time support provided by MetaFlex for the scripting application uses two of the features discussed in the architecture: 1) instance creation by name, and
2) function dispatching by name.

The Metaflex generated metaclass instance supplies instance creation by name, and the function dictionary supplies the function dispatching. An instance of class FuncDict provides the following services:

Lookup services. Function information can be looked up by exact key (the simple name and argument type list), by simple name alone, or by using a name that is comprised of the simple name and first part of the argument type list. For example, the key "Lookup(char*,i" would uniquely identify the second Lookup function in the list below. When there is ambiguity, it is possible to have a collection of all matching function members returned. For example, a call to the first LookupAll function below with "LookupAll" as the argument would return an OrderedCollection with FuncInfo instances for the two LookupAll functions below.

```
FuncInfo*              Lookup  (char*   pszNameNPartOfArgs);
FuncInfo*              Lookup (char* pszSimpleName,  int iArgCnt);

OrderedCollection*     LookupAll (char* pszWithThisSimpleNm);
OrderedCollection*     LookupAll (char* pszWithThisSimpleNm,int iArgCnt);

char*                  GetFullName (char* pszWithThisSimpleNm);
char*                  GetFullName (char* pszWithThisSimpleNm
                                        , int iArgCnt);
```

Function dispatch service. The function specified in the name key (2nd argument) is dispatched for the client to the instance of this class that "understands" this protocol (1st

argument). The number of arguments and their values are passed as the third and fourth arguments, respectively. The ArgV (4th argument) is an array of the arguments corresponding to the formal parameters of the function to be dispatched. The function dispatcher in the function dictionary uses this information to properly construct the stack frame for the function call.

```
void    *Execute (void *anInstanceOfThisClass
                  ,char *pszSimpleFuncName  // or pszNameNPartOfArgs
                  ,int iArgCnt
                  ,void **ArgV);
```

Each FuncDict contains a number of FuncInfo instances, one for each method that is part of the class being modeled with the Metaclass information. The following information is provided for each member function in an instance of class FuncInfo:

- Full function name (e.g. "foo(int, Ferrengi*, Bar&)"), a human readable version of the part of the member function declaration that is used by the compilers to generate unique ("mangled") names for a function.
- Simple name (e.g. "foo")
- Whether its virtual or non-virtual
- Its address (or virtual table offset)
- The number of arguments for this member function
- An encoded representation of the function's return type and arguments used to understand the length of things.
- The string representation of the return type of the function, that may be used to access its metaclass services (assuming it is a type for which a metaclass has been defined).

In this section, we have described the compile and run time responsibilities of the MetaFlex implementation to support extension of the scripting application. In the next section, we will evaluate our implementation, highlighting issues that may be helpful to others who may choose to implement a flexible metaclass generator service.

Evaluation

Without an automation tool like MetaFlex, we would not have been able to generally extend the scripting application in our application framework. Specifically, the ability to delegate the responsibility of responding to scripting requests (the Script Manager) and dispatching functions (to the function dictionaries generated for various application classes) from the system's application objects would not be possible. Without metaclass support, application objects would have to be explicitly knowledgeable about scripting, making them generally un-wieldy and less useful. With the MetaFlex generated support, these objects can be manipulated through scripts without compromising the original object semantics.

A number of key issues raised during our development are covered in the following sub-sections.

Specifying Metaclasses

In our early implementations, MetaFlex generated metaclasses for all classes in an application. To reduce the amount of generated code, a simple, but expressive scheme was needed for programmers to selectively choose which classes should have metaclasses generated for them by MetaFlex. Unfortunately, *simple* and *expressive* are competing goals.

With this in mind, we considered three ways for the engineers to specify their metaclass requirements:

1) Mark individual class declarations.
2) Mark a root class so that all derived classes from that root class will have metaclasses generated.
3) Specify an application's metaclass requirements in a separate file.

For the scripting application, the first approach was selected, primarily because this method did not explicitly interfere with the current development practices of the engineers, some of whom were already comfortable using a macro based metaclass solution for generating the vanilla metaclass functionality. The MetaFlex system is keyed on declarations these macros were generating in the expanded source input stream. Macros are difficult to maintain, however, compared to other specification methods. The macro approach, while capable of implementing the specification language we suggest, suffers from un-readability.

The second approach seems attractive, since it would require less effort to mark the classes that MetaFlex would act upon. It suffers, however, in that the metaclass specification for a class is isolated from it - in the declaration(s) of its parent class(es), which are typically found in a different file(s). Furthermore, there is a distinct possibility that all classes in the hierarchy would not necessarily require the same metaclass implementation. In our application framework, however, we do have a class hierarchy where this approach would work nicely. As a result, the metaclass specification language has support for this possibility.

The third approach, like the second, also suffers from the locality of reference problem. Unlike the first solution, it forces engineers to have explicit knowledge about the complexities of modeling type information, when all they really wanted to do was ask about what kind of object they have. More sophisticated means of expressing systems becomes available at the price of increased complexity. Being specified in a separate file is also attractive, in that one is able to manage the metaclass specification in one place, even though it describes classes that are defined elsewhere. For these reasons, this is the likely approach we will pursue if we continue to use our own MetaFlex tool to generate metaclasses for our applications' type systems.

Another dimension to the specification problem is whether a function dictionary should model all the methods in a class, once it has been determined that a metaclass should be generated. In the MetaFlex implementation we are currently using for scripting, meta-information for all methods in a class are generated and made accessible. This is a violation of the private and protected notions in C++. Ideally, access patterns should follow the same semantics as set forth in [Ellis90]. To support this variation, access control syntax has been added to the metaclass specification grammar described in the architecture section of this paper.

It ultimately may be desirable for engineers to explicitly choose which class members should be exposed for access by external applications. For example, for several classes in our scripting application, we would have liked to create function dictionary entries for only those methods that permit query access to end users. This not only reduces the size of the function dictionary considerably, but places further control over what an external scripting environment can do to internal application objects.

Type Checking

Better type checking from MetaFlex is desirable for the scripting application. After parsing end-user scripts, the Script Manager constructs the script parameters as an array of arguments of type *void** and invokes MetaFlex. MetaFlex, at present, checks to see the number of arguments in the supplied array and the number of arguments needed in the command initialization are the same. No additional checking is currently done, however, to see if the types match between each argument in the array and the corresponding type information of the function dictionary member to be dispatched. In order to make the system robust, better type checking is a must before command dispatching.

Type checking by MetaFlex can be done by querying the *isA* relationship of the arguments supplied in the array. For example, a command's initialization method declared as Init(Window*, Rct*, int) is supplied the array (void* val1, void* val2, void* val3). Since MetaFlex already maintains type information of the arguments in the command's metaclass, all it needs to do is to check that val1 isA Window and val2 isA Rct, before dispatching the command. If MetaFlex were to use this approach, however, it could only check the argument types that also have metaclasses created for them.

Name Overloading

To reliably disambiguate overloaded method names, the full specification of the name is often necessary for MetaFlex to be able to dispatch the correct function. Consider the example where there are two initialization methods of a class with the same number of arguments: Init(Window*, Foo*, Bar*) and Init(Window*, Foo*, FooBar*). In order for MetaFlex to select the correct method, the Script Manager could request MetaFlex to return both the initialization methods and then decide which method was appropriate, based upon information in its possession. Alternatively, the Script Manager could supply sufficient argument type names in its method lookup key to choose the appropriate method. At worst, all argument types need to be supplied.

In the context of component-based systems, possibly supplied by different software companies, the name disambiguation problem becomes even more complex. Without planning and cooperation, it is possible that classes in different components will be created with the same names. While this may be the intent in some situations, some orderly means of introducing new as well as replacement components is needed. Although we did not resolve this issue in our implementation, one suggestion we came up with is to assign a unique identifier to every command, possibly using an Internet-like addressing scheme. For example, a N-byte identifier might be used that identifies the software company developing the component, the component type, the command identifier, and version of the command. A scripting extension, like the one described in this paper, could then use this command identifier to dispatch the correct command. ISO and ANSI committees are actively investigating solutions to this complex problem.

MetaFlex Maintenance and Performance

As previously mentioned, our early implementation of MetaFlex generated metaclasses for all classes in an application. This initially did not pose a problem for the MPW compiler, but, caused both the MPW lib and linker utilities to choke as the generated amount of code exceeded their segment limits. To overcome these limits, we modified MetaFlex to automatically split the metaclass implementations such that no more than *X* number of them were put into any one code segment. Later, as the number of metaclasses grew, the compiler limits were exceeded as well. To manage this limitation, generated source was divided into files with no more than *Y* class implementations per file. Both of these values may be overridden at the invocation of MetaFlex.

With each revision of the C++ grammar, the change in its parser semantics needs to be updated in MetaFlex's parser - this is a challenging task. During our development of the scripting application, we noticed that component developers used C++ syntax that would not cause an error during regular C++ compilation but would break when we run it through the parser in MetaFlex. The reason is C++ compilers allow archaic C expressions not defined in ANSI C++ BNR forms. To accommodate these expressions, we had to fix the parser used by MetaFlex.

In general, performance was a drawback to our implementation of MetaFlex. At this writing, nearly seventy classes are having metaclasses with function dictionary dispatching capabilities prepared form them. MetaFlex generates approximately 620 K-Bytes of source code for them. This process, in turn, adds nearly thirty minutes to the build cycle for our script-aware application.

We have preliminary designs for several parser speed-ups that would mitigate this situation, but frankly, would prefer that the compiler vendors incorporate the ideas expressed in this paper into their compilers. If C++ compilers provided a MetaFlex-like

service, significant performance gains should be achieved, if for no other reason than the code would not have to be parsed twice.

Metaclass Specification Recommendation

In the final analysis, we would prefer that the compiler vendors implement a metaclass specification mechanism in their respective compilers that has all of the characteristics we have discussed here. For example, it would not be too difficult to imagine that the metaclass specification language, offered in this paper, could be implemented as a set of _keywords, or perhaps with the use of pragmas. These methods offer two ways of extending the C++ language. Both of these techniques potentially could be employed. In our metaclass specification grammar, nearly twenty tokens are introduced, and we don't think that the grammar is complete. Since it is difficult to have even one new language keyword adopted, it would be prudent to use the #pragma construct to implement most, if not all, of the metaclass specification.

The MakeMetaClassFor term in our grammar could be transformed into _MakeMeta keyword or #pragma declaration. For example, the first example in our specification language section, presented earlier, might be implemented as follows:

```
#pragma MakeMeta      Class DatabaseObject
                      Support FDict Style AddrsOnly
                      Install DBAccessPolicyManager
```

Our recommendation to compiler vendors is to allow developers to site metaclass specifications anywhere in the type system. Some developers will find it most attractive to co-locate their metaclass specification in the class being modeled. Others may want to co-locate all metaclass specifications in one file, as we suggested in the evaluation above. This may add some complexity to the problem (e.g., Which specification takes precedence when more than one are present in the type system for a given class? Do standard scoping rules apply?), but offers the greatest flexibility and customer satisfaction.

Conclusions

We have presented the need for a flexible metaclass generator to build extensions to applications. A number of extensions that would benefit from such a service were illustrated, including a detailed case study on a scripting application that is part of our component based application framework. Since the MetaFlex service is not commercially available in C++ compilers, we were forced to develop an in-house solution that served our needs. Through an evaluation of the MetaFlex generation service, we highlighted a number of issues we faced in our development and discussed the relative merits of possible solutions.

From our experience, it has become quite clear that it is impossible to build an ideal metaclass that suit the needs of all extensible applications. To mitigate this difficulty, we presented a metaclass specification language that allows software developers the means to engineer appropriate run-time type information, tailored to the application's needs. We recognize that the grammar presented is, by no means, a complete elaboration of what may be required, but does offer an organized view of the issues surrounding metaclass specification.

It is our belief that compiler developers should provide flexible metaclass generation capabilities to model an application's type system, and that vendors who provide such a service, will have a competitive edge for supporting the current trend of building extensible applications. It would be best if the development environments provided a graphical interface that allows developers to specify which classes, or roots of classes, should have metaclasses built for them.

Acknowledgments

We would like to acknowledge Roger Voss, Jim Murphy, and Krishna Uppala for their valuable critique of our paper. A special recognition goes to Krishna Uppala who played a principal role in the development of the C++ parser. We also recognize the valuable contribution that Jim Murphy made by introducing the notion of metaclass to the engineering teams at Aldus. Pat Ferrel, Erik Christiansen, Robert Meyer, Scott Moody, and Murugappan Palaniappan developed the component based application framework.

References

[Apple91] Apple Computer, Inc, *Inside Macintosh Volume VI*, Addison-Wesley, Reading, MA, 1991.

[Böcker90] Heinz-Dieter Böcker, Jürgen Herczeg, "What Tracers Are Made Of" *OOPSLA/ECOOP '90 Proceedings,* 21-25 October, 1990.

[Christiansen92] Erik Christiansen, Mark Cutter, Pat Ferrel, Robert Meyer, Scott Moody, Murugappan Palaniappan, "Platypus: Aldus Scalable Component Architecture," Aldus Technical Report (1992).

[Cointe87] Pierre Cointe, "Metaclasses are First Class: the ObjVlisp Model," *Conference Proceedings of OOPSLA '87,* October 4-8, 1987.

[Cox86] Brad Cox, *Object-Oriented Programming: An Evolutionary Approach,* Addison-Wesley, Reading, MA, 1986.

[Ellis90] Margaret A. Ellis, Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990.

[Ferrel89] Patrick J. Ferrel, Robert F. Meyer, "Vamp: The Aldus Application Framework," *Conference Proceedings of OOPSLA '89*, October 1-6, 1989.

[Goldberg83] Adele Goldberg, David Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

[Kleyn88] Michael F. Kleyn, Paul C. Gingrich, "GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views," *Conference Proceedings of OOPSLA '88*, September 25-30, 1988.

[Maes87] Pattie Maes, "Concepts and Experiments in Computational Reflections," *Conference Proceedings of OOPSLA '87*, October 4-8, 1987.

[Murray92] Robert B. Murray, "A Statically Typed Abstract Representation for C++ Programs," *Usenix C++ Conference Proceedings*, USENIX Association, August 10-13, 1992.

[Palay92] Andrew J. Palay, "C++ in a Changing Environment," *Usenix C++ Conference Proceedings*, USENIX Association, August 10-13, 1992.

[Pérez92] Edward R. Pérez, Moira Mallison, "Sentries and Policy Managers: Providing Extended Operations for Objects," Texas Instruments Inc, October 16, 1992.

[Richardson92] Joel Richardson, Peter Schwarz, Luis Felipé Cabrera, "CACL: Efficient Fine Grained Protection for Objects" *Conference Proceedings of OOPSLA '92*, Andreas Paepcke, ed. (1992).

[Stroustrup92] Bjarne Stroustrup, "Run Time Type Identification for C++," *Usenix C++ Conference Proceedings*, USENIX Association (1992).

[Voss92] Roger Voss, "Virtual Member Function Dispatching for C++ Evolvable Classes," Aldus Technical Report (1992).

[Voss93] Roger Voss, "C++ Evolvable Base Classes Residing In Dynamic Link Libraries," To appear in *C++ Journal*, Vol. 3, No. 1 (1993).

[Wells92a] David L. Wells, José A. Blakeley, Craig W. Thompson, "Architecture of an Open Object-Oriented Database Management System," *IEEE Computer,* Vol. 25, No. 10 (1992).

[Wells92b] David L. Wells, Moira Mallison, Edward R. Pérez, "Behavioral Extension Mechanisms in Open Object-Oriented Database System," Texas Instruments Inc. (1992).