RJ 9735 (84621) March 15, 1994
Computer Science

Research Report

MANAGING CHANGE IN THE RUFUS SYSTEM

Peter Schwarz
Kurt Shoens

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, California 9512-6099

Research Division
Almaden • T.J. Watson • Tokyo • Zurich

RJ 9735 (84621) March 15, 1994
Computer Science

MANAGING CHANGE IN THE RUFUS SYSTEM

Peter Schwarz
Kurt Shoens

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, California 9512-6099

ABSTRACT: Rufus is an information system that models user data with objects taken from a class system. Due to the importance of coping with changes to the schema, Rufus has adopted the conformity-based model of Melampus [Richardson 91a]. This model enables Rufus to cope with schema changes more easily than traditional class- and inheritance-based data models.

This paper reviews the Melampus data model and describes how we implemented it in the Rufus system. We show how changes to the schema can be accommodated with minimum disruption. We also review design decisions that contributed to streamlined schema evolution and compare our approach with those proposed in the literature.

1 INTRODUCTION

Recent years have seen a tremendous increase in the amount of information that is available on-line. While some of this information is stored in traditional database management systems, a far larger amount is stored in ordinary files, in an ever-expanding set of formats determined primarily by the applications that have been developed to manage this data. Several obstacles must be overcome before one can exploit the information contained in this large and loosely-organized body of data. First, one must be able to locate information of interest, and then one must be able to examine and retrieve it despite having little or no familiarity with the application programs that were designed to manipulate it. Solving these problems is the goal of Rufus [Shoens 93], a system designed to facilitate access to information stored outside of database management systems.

The environment in which Rufus operates is both more diverse and more fluid than the environment faced by a traditional database management system. Rufus must handle data from a wide variety of sources, and new kinds of information and new formats are frequently introduced, making it impossible to develop a comprehensive schema *a priori*.

This paper describes how Rufus was designed and implemented to operate in an environment characterized by such changes. We begin with an overview that describes how Rufus captures data to support querying and provides the ability to manipulate query results. Next we describe how we adapted the conformity-based data model of Emerald [Black 87] and Melampus [Richardson 91b] to represent user data in Rufus. In Section 4, we discuss the Rufus implementation of this data model. We conclude with a look at related work and future directions for Rufus.

2 OVERVIEW OF RUFUS

Much of the information stored on-line today is a mixture of structured and unstructured data. For example, consider a document produced with a markup language like LaTeX. Although the source form of the document is stored in an ordinary text file, the document has considerable structure: it has a title, one or more authors, various named and numbered sections, etc. Furthermore, the source file has a well-defined relationship to other files: a bibliography file, files containing subparts of the document, etc. The contents of the sections, however, are arbitrary text and can be considered as unstructured data.

The document described above is one example of what we refer to as *semi-structured* data, and one can envision many other examples. We are all familiar with electronic mail, which contains both structured data in header fields and unstructured data in the message body. Other kinds of information are not yet as widely available in electronic form, but fit the same model. For instance, a television program listing contains a structured description of the program's name, time, channel, stars, and so forth, plus an unstructured description of the program's content. Non-textual data can also often be characterized as semi-structured. The computer representation of an image, for example, typically contains some structured information about the image's size and colors, plus unstructured pixel values.

Rufus attacks the problem of finding information by providing the ability to query both the structured and unstructured content of files. By recognizing the structured aspects of the data, Rufus is able to support more powerful and selective queries than systems that treat files as uninterpreted streams of text or bits. Thus, Rufus can distinguish documents with "Jimmy Carter" in the title

from documents written by Jimmy Carter or those that merely mention "Jimmy Carter" in their body.

Rufus addresses the problem of exploiting information once it is found by providing the user with a suite of generic and type-specific operations for manipulating the data located by a query. Thus, common actions like *Print* or *Display* can be applied to almost any type of query result, and specialized actions like *Reply* or *Compile* can be applied where appropriate.

A central tenet of Rufus is that it is impractical to require users to give up the formats in which their data is currently stored. Moving the data from its current form into a database would require that all the existing application programs be rewritten to operate against interfaces provided by the database, rather than the files that they currently access. This would be a difficult process at best, and any incompatibilities between the application data model and the one supported by the database would exacerbate the difficulty. Instead, Rufus discerns the types of files using a scoring system based on keywords, file name patterns, and the presence of certain bit patterns near the beginning of the file [Shoens 93]. Rufus uses the file's classification to select an appropriate routine to *extract* structured values from the file and *index* the file's unstructured data. The result of this *import* process is a pair of databases: an object database containing extracted attribute values, and a text database containing word occurrences. Each piece of imported data is presented to the user as an object with methods that apply existing applications to the underlying data, return attribute values, or navigate among related objects. Users can issue queries against the text and object databases to locate interesting information.

Consider the following example. Suppose the user has a number of files pertaining to a particular piece of software. The files include the source and object code for the program itself, a "make" file (instructions for compiling the program), some supporting documentation, and various mail messages describing bugs, requests for new features, etc. As each of these files is imported by Rufus, it is classified, structured information is extracted, and unstructured information is indexed. The source code, for example, would be classified as a C program. Each function defined in the file would give rise to a corresponding object in the object database, containing information about the function's name, number of lines, number and type of parameters, etc. Variables occurring in the body of the function would be stored in the text database, as would any words found in comments in the code. Similarly, the mail messages would be classified as Mail. For each message, the object database would contain information about the message's sender, the date it was sent, and so on. All the words occurring in the message body would be indexed in the text database. The other kinds of files would be handled in a similar type-specific manner. Once the files were imported, one could pose queries like "Find all the files that were modified after June 1, 1992 and contain the string 'operating system bug.' " This query might return a heterogeneous collection of objects, including source code, documentation and mail. For each object returned by the query, Rufus would provide a menu of appropriate actions. All of the objects support actions like *Print*, and Rufus invokes the appropriate application in each case: a pretty-printer for the source code, the relevant formatter for the documentation, and a simple ASCII print procedure for the mail. Other actions would only be appropriate for specific object types, e.g. the *Reply* action would be available for the mail objects, and the *Compile* action would be available for the source code. All of these actions could be implemented using existing applications, because all the underlying files are retained in their original form. Thus, Rufus not only helps you to find what you are looking for, it also helps you to use what you find without requiring existing applications to be rewritten.

3 THE RUFUS DATA MODEL

As stated above, each piece of semi-structured data that has been imported by Rufus is presented to the user as an object with an associated suite of methods. For the most part, objects correspond to fairly large units of data that users think of as distinct. For example, a mail message is presented as a single Rufus object, rather than as a separate object for each line or word in the message. Therefore, methods map to familiar user-level operations.

Encapsulation of data along with its associated behavior is a fundamental principle of object-oriented programming, and our desire to help users both find and exploit information made an object-oriented data model a natural choice for Rufus. The object-oriented concept of *subtyping*, which recognizes that different kinds of objects share common behavior, matches well with our desire to support generic operations like *Print* or *Display* across a variety of kinds of data. Similarly, different kinds of data share common attributes, like *Author* or *Title*.

In a more narrow sense, we will also refer to the structured information extracted from a file during import as an object. These objects are stored in Rufus's object database, and their representations and methods are defined using the Rufus schema language, to be discussed in Section 4.1. Since these two kinds of objects are usually in one-to-one correspondence, this distinction is not always important. In what follows, we will generally use the term "object" to mean an object in the Rufus database, rather than in the more general sense.

The Rufus data model is a subset of the Melampus data model [Richardson 91b], which in turn was inspired by the Emerald programming language [Black 87]. The Melampus data model retains the key features of Emerald, namely a sharp distinction between abstract data types and implementations and an implicitly-defined type lattice. Melampus omits some of the more esoteric features of Emerald (e.g., types as first class objects and parameterized types). Rufus is the first implementation of the Melampus data model.

The remainder of this section reviews the subset of the Melampus data model used in Rufus.

3.1 Abstract Types, Implementations, and Conformity

An *abstract type* is expressed as a set of method signatures, and specifies an interface. Each object is an instance of an *implementation*, which specifies a representation for the object and code for its methods. A particular object *conforms* to an abstract type if its implementation supplies at least those methods in the interface specified by the type. Similarly, one abstract type conforms to another if the first type's interface includes all the methods required by the interface of the second type.

An example should give an intuitive feeling for this approach; a more formal definition of these concepts is given in [Black 87]. Consider the abstract type *Mail*. The interface for this type includes signatures for methods like *Reply*, *Forward*, and *Subject*. Contrast this abstract type with the implementation *RFC822Msg*, which supplies code to implement the methods *Reply*, *Forward*, *Subject*, and *Print* for a particular kind of electronic mail. Since objects instantiated from this implementation support all the methods specified for the type *Mail*, they conform to this type. They also, however, conform to the abstract type *PrintableDocument*, whose definition states that objects conforming to this type must supply a *Print* method.

For our purposes, the key advantage of a data model based on conformity is that the type lattice is implicitly defined. Class definitions in many other object-oriented data models, including those of C++ [Ellis 90], Trellis/Owl [O'Brien 87], ORION [Banerjee 87b], and IRIS [Fishman 87], explicitly state the subtype-supertype relationships between classes. In our data model, the relationship between two abstract types or an abstract type and an implementation is established by applying an algorithm that tests for conformity [Black 87]. The flexibility of this approach is apparent. The preceding example demonstrated how a single implementation could conform to multiple abstract types. Similarly, multiple implementations that conform to a given abstract type can coexist. For example, the implementation *BitNetMsg* might also conform to the *Mail* type. The appropriate implementation for each object is determined by the classifier.

It is important to note that conformity is a subtyping relationship, comparing two specifications of behavior while saying nothing about the mechanisms that are used to implement that behavior. The implementations *RFC822Msg* and *BitNetMsg* might be completely distinct, or might share common code via *inheritance*. To encourage reuse of code when building implementations, we introduce an inheritance mechanism in the Rufus schema language (see Section 4.1). In many other object-oriented databases and programming languages, subtyping and inheritance are tightly coupled.

Since the type lattice is implicit, it is a simple matter to define a new abstract type or implementation. Because the conformity algorithm, rather than a declaration, determines where the new type or implementation fits into the existing lattice, there is no need to modify other definitions to reflect the addition (or deletion) of a type or implementation. As we will show in Section 4, our implementation was carefully designed to preserve this independence.

In a diverse and changing environment, the ability to develop a schema incrementally is valuable. A basic Rufus system can be delivered that includes definitions for a few widely-used implementations and abstract types. An individual wishing to extend Rufus to support a new kind of data supplies an additional implementation that specifies an object database representation for extracted information, a constructor for creating a database object from an underlying file, and code for implementing appropriate methods.[1]

Coexisting versions of the same implementation can be supported. When a new version is created, existing objects continue to use the original version of the implementation, but instances of the new version can be created when additional objects are imported. Queries and other applications, which are written using abstract types rather than implementations, are completely unaffected by such changes.

Additional abstract types can easily be defined, perhaps to be used in specific queries. For instance, a user might incrementally define the abstract type *Authored* to describe any kind of object with an *Author* method, and then search all objects conforming to this type for those where the *Author* method returns a specific value.

In a distributed environment, the Rufus data model permits groups of abstract types and implementations to be developed independently, and subsequently exchanged on an as-needed basis. It

1. In practice, it is also necessary to modify the classifier so that it can recognize the new kind of data and create instances of the new implementation. The classifier is outside the scope of this paper.

is not necessary for a consistent global schema to be shared between multiple sites or multiple groups of developers.

3.2 Collections

Rufus provides persistent collections for grouping sets of objects. An object can be in many collections at the same time. Rufus itself uses collections to maintain an extent for each implementation. Queries range over collections, and the result of a query is a collection. Collections can also be explicitly constructed by users. Collections are objects in their own right, and can themselves be members of collections.

Collections in Rufus can contain objects with arbitrary types. The *intersection type* of all the collection members is derived from the set of the methods that all member instances support. Prior to query execution, Rufus verifies that all methods mentioned in the query are supported by all members of the collection. Queries can also be restricted to those members of a collection that conform to a particular abstract type.

Since collections can contain many members, they can be indexed to improve query performance. An index can be dynamically created by specifying the name of a method whose return value is to be used as the key. An indexed method must have no side effects, take no arguments (other than the *self* parameter), and depend only on the object's immediate state. If an object that does not support an indexed method is added to a collection, we treat the indexed value as NULL.

4 IMPLEMENTATION

Rufus is written in the C language and runs on Unix workstations. A complete Rufus installation consists of common infrastructure plus a set of object implementations tailored for the particular kinds of data to be imported. Major infrastructure components include the classifier, schema compiler, object database, text database, method dispatch mechanism, access methods, and query executor. This paper focuses on the schema compiler, which translates an implementation written in the Rufus schema language into dynamically loadable machine code, and the dispatch mechanism, which supports invocation of methods on objects stored in the object database.

Figure 1shows the Rufus system structure.

4.1 The Rufus Schema Language and Schema Compiler

The Rufus schema language is used to write object implementations. Each implementation corresponds to a particular kind of semi-structured data that Rufus can import and operate upon, and one or more objects are instantiated for each piece of data that is imported. An implementation includes definitions of the methods that its instances support, the names and types of the instance variables in the record used to represent instances in the object database, and the definition of the implementation's constructor, which extracts values from the underlying data and uses them to initialize the object's instance record.

Figure 2 shows the declarations of a sample implementation for RFC 822 format electronic mail. The **representation** section defines the fields of an instance record. Base types supported by the language include integer, string, real, and date. Typed object references, which are constrained to

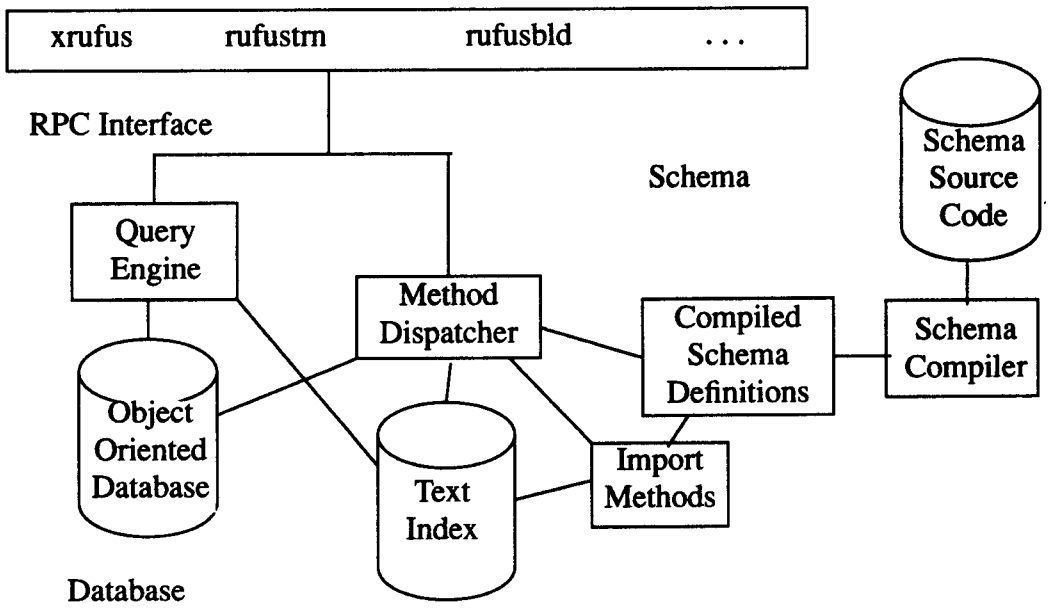

Figure 1. Rufus System Structure

point to objects supporting a particular set of methods, can also be declared. Record and sequence types are provided to construct complex types.

The **method** section gives the signatures of the methods defined by the implementation. In addition, methods are automatically generated to return the values of representation fields that are marked **public**. The executable code for a method is included after the **%code** marker. A constructor definition is similar to a method definition. Figure 3 shows an example of method code. The code is expressed in the C language, with a few extensions. These extensions are used to invoke other methods (**%call**), create new objects (**%new**), declare local variables with abstract types (**%var**), and discover the object ID of the *self* parameter (**%oid**). The schema compiler uses these constructs to track the abstract types of local variables and to type check method invocations. Since the C language does not support abstract types itself, it is possible for the programmer to subvert the type checking. Nevertheless, the schema compiler will detect inadvertent type errors.

The schema language provides a means of sharing both abstract type definitions and partial implementation definitions.Type definitions can be written into separate files with names that end in `.type`. For example, Figure 4 shows the definition of the *Mail* abstract type. The definitions in a type file are made known to an implementation via the **import** statement. In Figure 2, **import** is used to get the definitions of *Mail* and *Directory* types. Type files may import other type files to obtain additional definitions. To simplify the task of writing type files, the schema compiler will only read a particular type file once. Therefore, each type file can include other type files as needed, without the programmer needing to worry if that type file has been included somewhere else.

Partial implementations, containing definitions of methods and instance variables, are pulled into an implementation with the **inherit** statement. For example, in Figure 2 *RFC822* inherits from

```
implementation RFC822
{
        inherit "Text.impl";
        import "Directory.type";
        import "Mail.type";

        representation {
                public string  subject;
                public date    posted;
                public string  sender;
                public string  sequence recipient;
                       Mail    inreplyto;
                ...
        }

        methods {
                showhistory(instance self);
                ...
        }

        constructor(string filename, Directory dir);

%code;
        RFC822(string filename, Directory dir) { ... }
        showhistory(instance self) { ... }
%end;
}
```

Figure 2. A Sample Implementation's Definitions

Text. Inheriting an implementation copies all the method and instance variable definitions. The inheriting implementation is free to override the code that implements a method, but the signatures of methods may not be changed. Note that inheritance is provided as a tool for code reuse and does not directly structure the type lattice. Programmers are encouraged to write *mixins* [Stefik 86], small inheritable implementations that provide commonly useful function. Conflicts that arise from multiple definitions of instance variables or methods inherited from other implementations are considered errors in Rufus.

Inheritance also copies the C code that implements methods. The schema compiler renames inherited C functions, so that people writing implementations need not adopt contrived naming conventions to avoid function name conflicts. Any inherited invocations of such renamed functions are modified accordingly, so that the behavior of code is not altered by inheritance in unexpected ways. Note, however, that Rufus method invocations (as opposed to C function invocations) are not modified in this way. This is in keeping with the object-oriented principle of extended self-reference, which requires a method invocation to always invoke the most specific implementation of the designated method.

```
%code;
void
showhistory(instance self)
{
        %var Mail curmsg;

        curmsg = %oid self;
        while (curmsg != ObjNULL) {
                printf("%-10s %s\n",
                        %call(curmsg, sender), %call(curmsg, subject));
                curmsg = %call(curmsg, inreplyto);
        }
}
...
%end;
```

Figure 3. Sample method code

Implementations are stored in files whose names end in `.impl`. The Rufus schema compiler turns a `.impl` file into a C language source file, which is then compiled by the host system's C compiler. The result is a stand-alone module that can be dynamically loaded. This process is illustrated in Figure 5. Installation of a new implementation can be done concurrently with other activity, such as querying or importing new files. Rufus fits the new implementation and any associated abstract types into the existing type lattice, and saves a copy of the implementation module in the database so that it can also be dynamically loaded by future instances of Rufus.

Each compiled implementation module defines a function named `_schema_hook` that, when called, returns a pointer to a data structure that describes the implementation. The data structure describes the abstract type of the implementation, any additional abstract types used in its definition, the layout of its instance variables, and the addresses of the C functions that implement each of its methods. Thus, a compiled implementation module is a complete description of the implementation. Each implementation also contains a time stamp assigned by the schema compiler to identify its version.

Although the definition of an implementation may rely on definitions taken from type files or other implementations, once an implementation is compiled, it is independent from changes in

```
type Mail methods {
        subject(instance self) returns string;
        posted(instance self) returns date;
        sender(instance self) returns string;
        inreplyto(instance self) returns Mail;
        ...
}
```

Figure 4. Sample abstract type definition

other implementations until it is compiled again. In an environment characterized by frequent change, we preferred the safety and stability of this model, in which updates to program and definitional information only take effect when explicitly requested through recompilation. Our approach is similar to that of many programming languages and standard program-building tools can be used to trigger recompilation when the constituent files of an implementation are changed.

4.2 Method Lookup

When invoking a Rufus method on some target object, the runtime system must be able to locate the required method code, based on the implementation of the target object and the name of the method being invoked. In some languages, e.g. Smalltalk [Goldberg 83], method lookup is completely dynamic: the runtime system examines the class of the target object and possibly superclasses as well, until a class implementing the method is found, if one exists. At the other end of the spectrum, languages like C++ have enough information available at compile time that a unique index into a table of method addresses associated with each object (the *vtbl*) can be generated for each invocation.

The requirements for method lookup in Rufus lie between these extremes. The specific implementation of the target object is not known until runtime, and in fact may not even exist when the calling module is compiled, frustrating any attempt to determine a unique index for each method, as in C++. On the other hand, unlike Smalltalk, each Rufus implementation is self-contained, so there is no need to search for a method implementation. Furthermore, because the abstract types

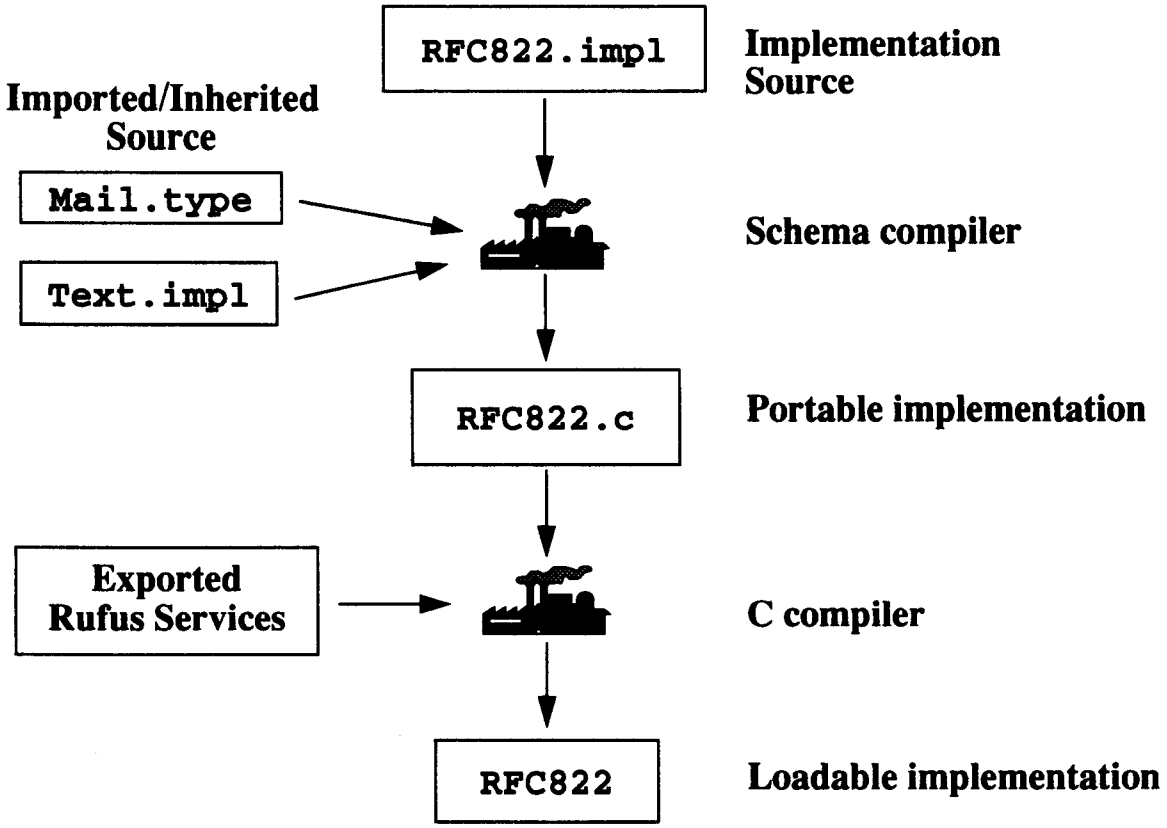

Figure 5. Schema compilation process

of variables are known to the schema compiler, invocations can be checked for type-correctness at compile time.

In Emerald, the problem of method lookup is solved by associating a translation table with each variable, and updating the table whenever an assignment to the variable is made. The table maps a method index determined by the variable's abstract type to the corresponding method address for the implementation of the object currently assigned to the variable. One such table must exist for each <abstract type, implementation> pair. This approach pays off if a variable is used as the target of several method invocations, as the cost of generating the vector at assignment is made up by the reduced the cost of each subsequent invocation.

In Rufus, different tradeoffs led us to chose a simpler implementation of method lookup. Each object instance contains an *implid*, an integer identifying its implementation. The address of a method is obtained by finding the implementation corresponding to the target object's implid and matching the method name against a table stored with the implementation. The table is sorted by name, making it amenable to binary searching. While this approach might appear to be prohibitively expensive, we think this will not be the case in practice. In the first place, the most primitive kinds of data (integers, strings and so forth) are not represented as objects in Rufus. Hence, we do not have to be concerned with method lookup for these performance-critical data types, which often require special implementation tricks in languages like Smalltalk.

Our second reason for choosing this implementation of method lookup stems from the way in which methods are used in Rufus. Methods are invoked in three scenarios: as a result of a user's request to perform some action on an object, during the execution of another method, and during evaluation of a query. Since the first case requires an interaction with the user for each invocation, such invocations are infrequent and not time-critical. One would expect the second kind of invocation to be dominant in a general-purpose programming language, but in Rufus, methods tend to be simple and the invocation of one method by another is relatively rare. Methods that correspond to user-requested actions like *Print* or *Display*, for example, in many cases simply act as a "bridge" between Rufus and some existing program to be applied to the underlying file. Other methods just return a field from the object's representation. Queries, however, may apply one or more methods to a large number of objects with a variety of different implementations. From a performance standpoint, therefore, the most important case to optimize in Rufus is the third one: invocation of a method from a query.

As noted in Section 3.2, queries range over collections. Our implementation of Collection objects currently allows any object to be inserted into any collection, but keeps track of which implementations are represented in each collection. During query execution, therefore, it is a straightforward matter to cache the result of a method lookup for a particular implementation, and use it later to apply the same method to another object in the collection with the same implementation.

If our simple approach to method lookup proves too costly in non-query invocations, a more general caching scheme could be implemented. The triple <calling implementation, signature number,[1] target implementation> is a unique key that can be used to cache the address of a target method. The first two components of this key can be provided by the schema compiler; the third

1. A signature number uniquely identifies an <abstract type, method name> pair within a given implementation, and is generated by the schema compiler when the implementation is compiled.

would be supplied at runtime by the target object. Subsequent method invocations with the same key could be quickly detected and the method address supplied from the cache.

4.3 Managing the Type Lattice

Although the type lattice can be developed incrementally, it is necessary to have an explicit representation of the current lattice available at runtime. The lattice is used to type check queries, to perform conformity searches (queries for which the collection to be searched is the set of objects conforming to some abstract type) and for query restrictions (queries that are restricted to those members of a collection that conform to a specified type). It is also convenient for the user to be able to browse the type lattice, to aid in formulating queries or exploring the space of available information.

All of the type information available to Rufus is contained in the executable implementation files produced by the schema compiler. We discarded the approach of building the type lattice at start-up by loading all the known implementations, because this approach clearly does not scale to a large number of implementations. Implementations can be large, and loading them into the address space when they may not be used is slow and wasteful. Instead, we opted to represent types, implementations, and the type lattice as persistent objects in the Rufus object database. Updates to the type lattice are implemented as methods on TypeLattice objects, which reflect incremental updates to the database. Having made this decision, we could, in principle, have implemented all the conformity checking and lattice navigation operations as Rufus methods invoked through our standard method dispatch mechanism. This approach, however, would have required a large number of method invocations for each conformity check or lattice navigation operation. Our compromise was to use this persistent representation of the type lattice to generate a memory-resident representation at start-up. The cost of building the resident lattice from the Rufus objects is much smaller than the cost of building it from the implementation files, and should allow Rufus to accommodate a large number of types and implementations without difficulty.

Building the implementations for these basic kinds of objects was an interesting and early experience in using the Rufus schema language. The persistent representation of a set of abstract types, in particular, requires a network of mutually referential objects to describe the abstract types in method signatures. Our success in building these implementations gave us confidence that the schema language was sufficiently powerful to build the types and implementations that would be required to represent objects derived from user data.

4.4 Reacting to Schema Changes

In the Rufus object database, all information about an object's behavior is contained in its implementation. As described earlier, the compilation process for an implementation may draw on other abstract type and implementation definitions. However, changes to an implementation are only made effective when the implementation is re-installed in the database. This section describes the process of installing new or modified implementations into the database.

The following types of change to a Rufus schema can occur: 1) adding a new implementation; 2) changing an implementation's interface or stored representation; 3) modifying an implementa-

tion's method code without changing anything else; and 4) deleting an implementation. Note that while these changes might be caused either by direct changes to an implementation definition or by changes to implementations that it inherits, the source of the change is irrelevant.

Rufus maintains two persistent tables for looking up implementations. The first table, called the name table, maps an implementation name to the most current definition for that name. This table is used to find the right implementation when a new instance of a named implementation is being created and is updated when new or revised implementations are defined.

Each version of an implementation is assigned an implid which is stored in its instances. The second table, called the implid table, maps implid's to implementation modules and is used to find the right implementation for object instances as described in Section 4.2.

All the schema changes described above can be accommodated by a small set of procedures. Cases 1) and 2) are handled by assigning a new implid to the new or revised implementation. This implid is recorded in the implementation's name table entry, so that when instances of the implementation are created they will bear the new implid. The implementations of existing objects are not disturbed; their implid's continue to point to their original implementations, which exist as long as there are corresponding object instances in the database.

Case 3) is handled by replacing the existing implementation with the new implementation without changing the name or implid tables. The change thus affects all instances of the implementation immediately. Case 4) is handled by deleting the name table entry for the implementation. When all instances of the implementation have been deleted, the implementation can be deleted from the implid table and the implementation module can be discarded.

Since users specify desired objects by *type* rather than *implementation*, they are shielded from irrelevant schema changes. For schema changes that affect the stored representation but leave the interface alone, users will perceive no distinction between new and old versions of the implementation. If the interface (set of method signatures) for an implementation is changed, users that do not depend on the changed method signatures will be able to operate on objects of both the old and new implementations. Users who care about the changed interface will discern the difference, because new instances will not conform to the desired type.

5 RELATED RESEARCH

Existing work in schema evolution has focused on understanding a change made to a class definition, propagating the effects of the change to the rest of the schema, and applying the resulting schema changes to the existing database [Banerjee 87a, Penney 87, Skarra 86, Zicari 91, Morsi 92]. These systems develop a set of rules governing the schema, then apply these rules after changes are made to return the schema to a consistent state, if possible, or to reject the changes otherwise. They then either convert existing object instances gradually ("screening") or all at once ("conversion").

None of these systems allow multiple versions of a class to be visible to an application. ORION [Banerjee 87a] comes closest, by supporting multiple versions of the schema. Even in ORION, however, each application must choose a specific schema version through which to view the database. Furthermore, none of these systems appear well suited to the task of merging independently-

developed sets of class definitions, since the addition of new classes can cause significant upheaval to the schema.

In our view, these problems are caused by the tight binding between inheritance and subtyping in these systems which leads to an explicitly-defined class lattice. Any change to a class requires that the entire lattice be adjusted accordingly. Thus, changes to a single class can affect many others. In addition, the requirement that each class have a single definition means that changes in the schema must be reflected, sooner or later, into changes in object instances.

By contrast, a compiled Rufus implementation is independent of the presence or absence of other implementations. The structure of the lattice is implicit and maintained by the system; individual types and implementations are placed in the lattice as determined by the rules of conformity. Rufus users specify data types in terms of method names and signatures rather than by naming classes. Thus, there is a separation between types (as specified by required methods) and implementations (a specific set of method and storage representation definitions). Rufus can allow object instances created by multiple versions of the same implementation to co-exist because the users of these instances are guaranteed to get objects that support the desired methods.

6 FUTURE WORK

Future work on Rufus will incorporate the notion of *aspects* [Richardson 91a], which allow an object's behavior to depend on context and to evolve over time. A Rufus object could have multiple aspects, each exporting a different set of methods. New aspects of an object can be created as needed, while the behavior of existing aspects remains unchanged. Aspects would be used to model context-dependent behavior. For example, consider a PostScript file. In most contexts, one would expect Rufus to present this data to the user as an object that conforms to the abstract type *Image*, with an interface including methods like *Scale*, *Rotate*, *Display* and *Print*. Situations may arise, however, when it would be preferable to view the data as an ordinary text file.

A problem that can arise when merging schemas is that independently-developed abstract types (or implementations) may use different signatures to represent operations that are conceptually very similar, if not identical.[1] As a result, two types (or an implementation and a type) that "should" conform may not, in practice. For example, an object conforming to one type may not conform to another because a method was inadvertently given a different name, or its arguments were specified in a different order. Such minor schema differences can be patched by creating an aspect conforming to the desired target type. The aspect's implementation invokes the method with the corresponding name or reorders the arguments.

7 CONCLUSIONS

This paper has described the implementation of the Melampus conformity-based data model in Rufus. Our design allows a variety of schema changes to be implemented with simple changes to system tables and without perturbing existing object instances. Schema changes also have minimal effects on applications and users. The key features of the model and its implementation are:

---

1. The inverse problem, accidental conformance of disparate types, does not seem likely to occur in practice.

1. Separation of subtyping and inheritance. The lattice describing how objects behave is not constrained by the structure of the lattice describing how objects are implemented.
2. Separation of type and implementation. Users don't specify data types by naming classes; instead they say what methods they care about.
3. No requirement that object instances correspond to the most recent version of an implementation. Existing instances need not be modified when new versions are defined and references to existing instances are similarly unaffected.
4. Self-contained implementations. Since a compiled implementation contains a copy of all inherited method code, implementations are completely independent. Rufus can easily keep copies of outdated implementations as long as there are corresponding object instances.
5. A programming language model of implementation definition. All of the dependency tracking gets taken care of outside the database, using conventional programming tools.
6. A method dispatch mechanism that permits the calling of methods when only the abstract type of an object reference is known at compile time.
7. A persistent representation of type lattice that scales to many types and implementations.

These features provide the flexibility that enable Rufus to be easily extended to handle new kinds of information.

Acknowledgments. The authors gratefully acknowledge the guidance and insights of Rakesh Agrawal and Joel Richardson.

REFERENCES

[Banerjee 87a] Banerjee J. and Kim W., *Semantics and Implementation of Schema Evolution in Object-Oriented Databases*, Proceedings 1987 SIGMOD Conference.

[Banerjee 87b] Banerjee J. and Kim W., *Data Model Issues for Object-Oriented Applications*, ACM Transactions on Office Information Systems 5(1), January 1987.

[Barbara 92] Barbara D., Clifton C., Douglis F., Garcia-Molina H., Johnson S., Kao B., Mehrotra S., Tellefsen J., and Walsh R., *The Gold Mailer*, Proceedings 1992 IEEE Data Engineering Conference.

[Black 87] Black A., Hutchinson N. Jul E., Levy H., and Carter, L., *Distribution and Abstract Types in Emerald*, IEEE Transactions on Software Engineering SE-13(1), January 1987.

[Davison 92] Davison W., *TRN–Threaded Read News Program*, 1992.

[Ellis 90] Ellis M. and Stroustrup B., *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[Fishman 87] Fishman D. et al., *IRIS: An Object-Oriented Database Management System*, ACM Transactions on Office Information Systems 5(1), January 1987.

[Gifford 91] Gifford D., Jouvelot P., Sheldon M., and O'Toole J., *Semantic File Systems*, Proceedings SOSP Conference, 1991.

[Goldberg 83] Goldberg A. and Robson D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Publishing Company, 1983.

[Goldberg 92] Goldberg D., Nichols D., Oki B., Terry D., *Using Collaborative Filtering to Weave an Information Tapestry*, Communications of the ACM, 35(2), 1992.

[Kim 88] Kim W. and Chou H.-T., *Versions of Schema for Object-Oriented Databases*, Proceedings 14th VLDB Conference, Los Angeles, 1988.

[Lerner 90] Lerner B. and Habermann A. N., *Beyond Schema Evolution to Database Reorganization*, Proceedings 1990 ECOOP/OOPSLA Conference.

[Malone 87] Malone T., Grant K., Turbak F., Brobst S., Cohen M., *Intelligent Information-Sharing Systems*, Communications of the ACM, 30(5), May 1987.

[Morsi 92] Morsi M., Navathe S., and Kim H.-J., *An Extensible Object-Oriented Database Testbed*, Proceedings 1992 IEEE Data Engineering Conference.

[O'Brien 87] O'Brien P. and Halbert D., *The Trellis Programming Environment*, Proceedings 1987 OOPSLA Conference.

[Penney 87] Penney, D and Stein J., *Class Modification in the GemStone Object-Oriented DBMS*, Proceedings 1987 OOPSLA Conference.

[Richardson 91a] Richardson J. and Schwarz P., *Aspects: Extending Objects to Support Multiple, Independent Roles*, Proceedings ACM SIGMOD Conference, 1991.

[Richardson 91b] Richardson J. and Schwarz P., *MDM: An Object-Oriented Data Model*, Proceedings Third International Workshop on Database Programming Languages, 1991.

[Shoens 93] Shoens K., Luniewski A., Schwarz P., Stamos J., and Thomas J., *The Rufus System: Information Organization for Semi-Structured Data*, Proceedings 1993 VLDB Conference.

[Skarra 86] Skarra A. and Zdonik S., *The Management of Changing Types in an Object-Oriented Database*, Proceedings 1986 OOPSLA Conference.

[Stefik 86] Stefik M. and Bobrow D., *Object-Oriented Programming: Themes and Variations*, AI Magazine, January 1986.

[Zicari 91] Zicari R., *A Framework for Schema Updates in an Object-Oriented Database System*, Proceedings 1991 IEEE Data Engineering Conference.