2000

A Model for Compound Type Changes Encountered in Schema Evolution

Barbara Staudt Lerner
University of Massachusetts - Amherst

A Model for Compound Type Changes Encountered in Schema Evolution

BARBARA STAUDT LERNER
University of Massachusetts, Amherst

Schema evolution is a problem that is faced by long-lived data. When a schema changes, existing persistent data can become inaccessible unless the database system provides mechanisms to access data created with previous versions of the schema. Most existing systems that support schema evolution focus on changes local to individual types within the schema, thereby limiting the changes that the database maintainer can perform. We have developed a model of type changes incorporating changes local to individual types as well as compound changes involving multiple types. The model describes both type changes and their impact on data by defining derivation rules to initialize new data based on the existing data. The derivation rules can describe local and nonlocal changes to types to capture the intent of a large class of type change operations. We have built a system called Tess (Type Evolution Software System) that uses this model to recognize type changes by comparing schemas and then produces a transformer that can update data in a database to correspond to a newer version of the schema.

Categories and Subject Descriptors: H.2.m [**Database Management**]: Miscellaneous; H.2.3 [**Database Management**]: Languages—*Database (persistent) programming languages*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Persistent programming languages, schema evolution

1. MOTIVATION

Databases frequently have long lives. During a database's lifetime, the database schema is likely to undergo significant change as new demands are placed on the data. The database schema serves two purposes. First, it defines an interface for programs and users to query the data contained

This work was supported in part by the Air Force Material Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under contract F30602-94-C-0137 and in part by National Science Foundation grant CCR-9504170.
Author's address: Department of Computer Science, Williams College, Williamstown, MA 01267; email: lerner@cs.williams.edu.

within the database. Second, it determines how the database management system physically stores the data on the disk. When the schema is changed so that the data can be used for a new purpose, this also impacts the way data is physically stored. The goal of schema evolution research is to allow schema definitions to change while maintaining access to data that has already been stored to disk.

There are two major issues involved in schema evolution. The first issue is understanding how a schema has changed. The second issue involves deciding when and how to modify the database to address such concerns as efficiency, availability, and impact on existing code. Most research efforts have been aimed at this second issue and assume a small set of schema changes that are easy to support, such as adding and removing record fields, while requiring the maintainer to provide translation routines for more complicated changes. As a result, progress has been made in developing the backend mechanisms to convert, screen, or version the existing data, but little progress has been made on supporting a rich collection of changes. The purpose of this work is to enrich the collection of changes supported, independent of the backend mechanism used to manage the data.

Existing database systems that provide schema evolution support changes isolated to individual types within a schema, such as adding a field to a record. More radical changes of representation, such as combining two records are either difficult or impossible with existing database systems. Changes isolated to individual types are not always sufficient, however. A new record may be created to combine the information of several related records, or a large record may be decomposed into several simpler ones. Such changes will clearly impact the representation of persistent data.

The flexibility in data type definition offered by object-oriented databases and persistent programming languages admits the possibility of more complicated changes than those typically encountered in relational database systems, making schema evolution a more difficult problem.

With persistent programming languages the evolution problem is more pervasive than with databases. When using a database, those types that have persistent data are defined in the schema, while transient types are defined in traditional programming languages that interoperate with the database. The transient types can be changed without impacting the persistent data. With persistent programming languages, there is typically no distinction in the programmer's eyes between transient and persistent types. In particular, some persistent programming languages, such as PGraphite, Pleiades, Napier-88, and PS-Algol [Wileden et al. 1988; Tarr and Clarke 1993; Dearle et al. 1989; Atkinson et al. 1983], treat persistence orthogonally to types. With these languages, an instance of any type can be made persistent dynamically. This approach is very powerful and flexible, since it allows programs to manipulate data uniformly without being concerned about whether it is persistent or transient data. It aggravates the evolution problem, however, because every type potentially has persistent data associated with it. Modifying any type definition can make some

persistent data inaccessible. While Java does not support orthogonal persistence, Java's serialization mechanism [Arnold and Gosling 1998] makes it trivial to turn any type into a type that can become persistent. Thus, while one can distinguish serializable types from non-serializable types, there is no need to change a type in any significant way to make it serializable. The result is that it is common for many types in a Java program to be serializable.

Unfortunately, there is little published data [Garlan et al. 1994; Sjøberg 1993] about how persistent or transient types change during maintenance. Researchers studying maintenance of object-oriented hierarchies, which are not necessarily persistent, cite modifying types in the hierarchy and reorganizing the hierarchy as frequently desirable activities [Johnson and Opdyke 1993 ; Opdyke and Johnson 1993; Lieberherr et al. 1991; Opdyke and Johnson 1990; Casais 1990; Mellor and Shlaer 1992]. One can expect, however, that maintainers are reluctant to make radical changes to an object-oriented hierarchy or any other persistent type or schema definition if those changes make it difficult or impossible to access existing data. As a result, the maintainers may sacrifice other desirable properties for their schemas and type definitions such as appropriateness of abstractions, modularity, efficiency, etc.

Our goal is to facilitate schema evolution involving complex type changes to allow more natural evolution of persistent types. We have developed a model of type changes incorporating changes local to individual types as well as compound changes involving multiple types. The model describes both type changes and their impact on data by defining derivation rules to initialize new data based on the existing data. The derivation rules can describe local and nonlocal changes to types to capture the intent of a large class of type change operations.

2. OVERVIEW

One way to design a schema evolution system is to define schema modification commands to implement each type change that we want to support. The advantage of this approach is that the maintainer can explicitly inform the schema evolution system of the changes. For each type change, the system defines the effect that the change will have on the data. By choosing the appropriate commands, the maintainer simultaneously modifies the schema and develops a transformer to update the existing data. For example, there may be a command to add fields to a record whose effect on the data is to create a new field set to a default value. Another command may be used to delete a field from the type. Its effect is to delete the data corresponding to that field. In this type of system, there will be many specialized commands to accomplish all the supported type changes such as changing the size of an array, adding a value to an enumerated type, etc. When dealing with type changes isolated to individual types, it is possible for us to enumerate all the changes that may occur, provide a command for each of these, and define precisely what the effect on existing data will be.

Now, let's consider a more complex type change. Suppose the maintainer wants to move a field from one record to another. If the maintainer applies the *delete-field* command on the original type followed by the *add-field* command on the new type, this will be treated as two separate commands. The semantics of the *delete-field* command will result in deletion of the associated data. The semantics of the *add-field* command will result in addition of a new field set to some default value. To solve this problem we could introduce a new command, *move-field*. Now, there are two ways in which the maintainer can modify the types. If one examines the type definitions after the changes using the two approaches, the definitions are identical. The effect on the persistent data is quite different, however. When using the *delete-field* and *add-field* commands, data is lost. When using the *move-field* command, data is preserved. The maintainer needs to understand this difference and needs to be careful in choosing which commands to use when modifying the types. The problem is that the command approach focuses on the editing *process* rather than the editing *result*. Furthermore, the number of commands would proliferate and the complexity of using the schema evolution system would increase as more complex type changes are supported.

Another alternative is to allow the maintainer to modify the types as necessary and then compare the two versions of the schemas to identify the changes, thereby focusing on the editing result rather than the editing process. The advantage is that the maintainer can edit the schemas using a normal editor, focusing on producing the correct new type definitions without worrying about the exact process used to create those new definitions. The disadvantage is that the system must now infer how the types have changed instead of being explicitly told. In this research, we have developed algorithms to perform these inferences by comparing successive versions of a schema to identify the changes. The schema comparison algorithms use naming similarities, structural similarities, and interrelationships among the types from successive versions to infer the type changes. Experimentation with these algorithms has demonstrated that they can identify a wide variety of type changes successfully.

As another example of a type change that requires understanding the impact on multiple types simultaneously, consider adding a new type to a schema. In most database systems this change is understood in isolation from other changes. When a new type is added, it has no impact on existing data. It is quite likely, however, that the addition of a new type actually represents the reorganization of other types in the schema. For example, an individual type may be split into two types. The desired effect on the data is to move the data associated with the fields of the new type from objects of the old type to new objects. We could define a *split-type* command to accomplish this, further complicating the maintainer's job. Instead, we develop algorithms to recognize that a type has been added and then look for modified types that may act as sources of information for objects of this new type. Similarly, in most database systems, type deletion results in deletion of objects of the type. Instead, our algorithms look for other

modified types that may serve as destinations for the data that would otherwise be deleted.

Of course, it is possible that the algorithms will make incorrect inferences. As a result, it is important for the maintainer to be involved in the type comparison process. We have incorporated maintainer control into the type comparison algorithms in several ways. First, the maintainer can control which types are compared, if desired. Second, the algorithms can generate multiple inferences of observed type changes from which the maintainer can choose. Third, the algorithms associate a qualitative assessment with each inference indicating the complexity of the change and its impact on the data. The maintainer can use these assessments to set thresholds on the comparison algorithms or to focus attention on the more complex inferences or those with greater impact on the data. Finally, the maintainer can ignore the inferences generated by the algorithm and explicitly tell the system what the impact on the data should be. Using these techniques the maintainer can guarantee that the type changes have the appropriate impact on the existing data. Note that this ability to review the anticipated impact on existing data is useful to remove ambiguity even if schema modification commands are used, particularly in the case where there are multiple commands that lead to the same result as in the *move-field* example earlier. The questions that must be addressed when evaluating the type comparison approach are the following. How difficult is it for the user to make type changes that occur most frequently? How difficult is it for the user to make complex type changes? Our experimentation indicates that type comparison algorithms can reliably identify typical type changes and can often generate correct derivation rules for complex changes.

When supporting changes local to an individual type, the appropriate object changes can be performed by modifying each object in isolation. For example, if a field is deleted from a type, each object of that type can be modified independently of all other objects. The same is not true when supporting changes that affect multiple types. Implementing a single change may require modifying more than one object. Consider moving a field from one type to another again. To implement this change correctly, we must move data from one object to another. This implies that we must identify pairs of objects to operate on. Our algorithms identify collections of objects in two ways. Objects may be related *structurally*. That is by dereferencing fields of one object transitively we may reach other objects that we need to modify. Alternatively, objects may be related by having a common *value*. This is similar to a relational join operation. Identifying these collections of objects is key to being able to implement complex type changes.

Most schema evolution research has addressed the problem of how to update existing data efficiently assuming the type changes are well understood. The emphasis of the research described in this paper is to understand how schemas change during evolution and to develop algorithms that can recognize those changes. Our goal is to represent the schema changes

that occur in such a way that their effect on existing data can be accomplished using a variety of data translation mechanisms. For example, in small databases that may belong to an individual user, we can make the database unavailable temporarily and transform all data in the database at once. For large, shared databases, we can employ more sophisticated algorithms such as those developed by Ferrandina et al. [1994] to transform individual objects as they are accessed in order to maintain high availability. In situations where the data is shared by many programs, schema changes may also impact a great deal of code. In those cases, we can use the inferences we produce to define views on the data. Thus, the emphasis of this research is to develop algorithms that can recognize complex type changes made by a maintainer. Instead of constraining the maintainer to perform only supported type changes using a small set of primitive type change commands, we give the maintainer great flexibility in how to change the types. We are addressing the front-end problem of understanding schema changes in a flexible manner to allow integration with a variety of data translation mechanisms.

The remainder of the paper is organized as follows. In Section 3 we describe related work in schema evolution. In Section 4 we present the type model used in this research. In Section 5 we describe a model of how data changes in response to schema changes. In Section 6 we present a model for simple type changes. In Section 7 we present our model of compound type changes. In Section 8 we present an example schema evolution that uses compound type changes. In Section 9 we describe some algorithms developed to compare schemas. In Section 10 we describe the prototype implementation. In Section 11 we summarize the results of our experimental evaluation. In Section 12 we describe future directions of our research and conclude in Section 13.

3. RELATED WORK

The problem of schema evolution was first addressed with respect to traditional database systems. While many database systems support a few simple changes automatically, such as adding or deleting record fields, only a few systems [Shu et al. 1975; Navathe 1980; Shneiderman and Thomas 1982] support more general transformations. In these cases, the maintainer is responsible for explicitly describing how to convert the data from its old format to its new format using a special purpose data translation language. This approach is a powerful one, but creation of the transformer is a manual process.

More recent database systems generate transformation functions based upon the changes made to the type definitions. Orion [Banerjee et al. 1987; Kim and Korth 1988] and GemStone [Penney and Stein 1987] are two object-oriented database systems that provide some evolution support. In these systems, evolution is defined in terms of primitive operations that change individual type definitions, such as adding instance variables to a class, removing instance variables from a class, and renaming instance

variables. Some type changes are completely automated, but at the expense of limiting the ways in which a maintainer can change type definitions. For example, in Orion the type of an instance variable can only be replaced by a supertype in the type hierarchy. More complex type changes, such as combining two records, are not supported directly. Instead this change is accomplished as several independent changes. The maintainer deletes each instance variable individually from one of the types. The maintainer adds an equivalent instance variable to the second type for each instance variable deleted from the first type. The maintainer modifies all references to the first type to refer to the second type.[1] Finally, the maintainer deletes the first type. Since each change is treated individually rather than as a collection of related changes, deleting the instance variables results in deleting the data contained in those instance variables. To preserve the data, the maintainer must develop code to move the data explicitly. In GemStone the maintainer directly extends the transformer, while in Orion the maintainer must develop and execute programs to move the data prior to deleting the instance variables containing the data.

Another approach to schema evolution relies on the simultaneous maintenance of multiple versions of types and data [Skarra and Zdonik 1986; Clamen 1994; Bratsberg 1992; Monk and Sommerville 1992; Tresch and Scholl 1992; Lautemann 1997a, 1997b; Ra and Rundensteiner 1994]. With these approaches, multiple versions of the same type exist within a single database. The advantage is that old and new code can operate on old and new data without requiring either to be changed. The disadvantage is that the maintainer must provide routines to make data appear to be of the version of the type that the code is expecting. This approach admits more general changes, but it still limits changes to be isolated to individual types. It also results in significant overhead (in both space and time) for maintaining and accessing multiple type and data versions. Odberg [1994] extends the versioning approach to the entire schema, which is versioned when a type is modified. This allows the description of changes that simultaneously affect multiple types, but still requires the maintainer to define the translation routines between versions.

O_2 [Breche et al. 1995; Breche 1996; Ferrandina and Lautemann 1996] is an object-oriented database system that supports evolution through an interesting integration of transformation and maintenance of multiple views. They minimize the number of schema versions created by categorizing schema changes into one of three categories: schema extending, compilation safe, and compilation unsafe. Schema extending changes have no impact on existing applications. Compilation safe changes require applications to be recompiled, but guarantee that compilation will be successful. Compilation unsafe changes might require modifications to applications in

[1]Note that if the type being deleted is used as the type of an instance variable, we cannot in general replace its type with the second type since the second type is not necessarily a supertype of the first type. In that case, we would need to delete the instance variable and create a new instance variable of the desired type.

order for them to compile. When schema extending or compilation safe changes are made, transformation functions are generated to translate the persistent objects. Since the applications do not need to change, it is beneficial to have one version of the objects that all applications share. When compilation unsafe changes are made, however, they create a new schema version and allow each application to work with the version that they compile against. They generate conversion functions that allow objects to be accessible through multiple versions of the schema. O_2 also provides higher-level operations to manipulate the class hierarchy better than previous systems. The high-level operations are defined as a composition of primitive operations. As a result, they provide better support for the maintainer in expressing type changes and preserving data. For example, the abstraction-generalization operation can be used to create a new superclass that generalizes a set of existing classes. While these high-level operations support more complex changes than previous systems, defining type changes via a predefined set of operations necessarily restricts the kinds of type changes that are supported. In particular, while they have numerous operations to allow the definition of new classes and migration of existing objects to these new classes, none of their operations allow simultaneous modification of multiple types such as moving a field from one existing type to another.

TransformGen [Garlan et al. 1994] is a system to support evolution of abstract syntax grammars used by Gandalf programming environments [Habermann and Notkin 1986; Habermann et al. 1991]. The abstract syntax grammars are analogous to type definitions; they define the format of the abstract syntax trees stored in databases maintained by Gandalf environments. The abstract syntax changes for which TransformGen automatically generates transformation routines are analogous to the type changes supported by Orion and GemStone. TransformGen goes beyond these two systems, however, by allowing the maintainer to modify the generated transformation using a declarative data manipulation language. In this way, the maintainer can perform complex type changes using the primitive operations provided and then easily fix the generated transformations to have the intended effect. The significance of this extensibility is that the resulting transformers can handle arbitrary type changes, including those involving multiple types, but without requiring the maintainer to write transformation routines. While the maintainer can extend the transformer, there is little guidance in identifying the limits of the generation process and the situations that require extension. OTGen [Lerner and Habermann 1990] is a system designed using the concepts developed in TransformGen to support flexible transformation of object-oriented databases. As such it has many of the features and limitations of TransformGen, but is aimed at a more general type system.

In contrast to existing database systems, we support evolution via type comparison algorithms rather than editing commands. In essence, we are performing a smart differencing of versions. This is substantially different

than the edit distance algorithms typically used to detect differences as used in Unix *diff* or spell-checking algorithms [Hall and Dowling 1980; Kukich 1992; Peterson 1980]. These algorithms operate on a string representation and support the four operations of substitution, transposition, insertion, and deletion. The algorithms are not robust with respect to ordering changes. Also, they result in identifying changes in terms of string differences which do not assist in dealing with the semantics of the changes. Changes to comments bear equal weight to changes in type definitions. Also, the difficult task of generating transformation routines based on the changes that occurred requires understanding the semantics of the changes, not just the syntax of the changes as differencing algorithms do. The algorithms presented here can identify the semantics of the changes because they perform the differencing by examining the semantics of the type definitions rather than simply their syntax. Substitution, reordering, insertion, and deletion are just some of the changes identified by the type comparison algorithms presented here.

4. TYPE MODEL

Before discussing the details of the type change model, we must first present the type model that we use. The type model is a language-independent type model that captures features common to many programming languages. The type comparison algorithms operate on types defined in this language-independent model. They assume that a translator can translate from the types of a specific programming language to the language-independent type model, thus providing reuse of the algorithms across a range of languages. We are concerned with the structural aspects of the type model as those are most relevant to understanding the impact of schema changes on persistent data. As a result, we do not treat the types as abstract types, although they may, in fact, be implemented abstractly. In our examples, we therefore present the type representations used, but not the interfaces or operations belonging to those types.

A schema consists of a collection of type definitions. Schema changes are performed by editing types within the schema. Editing a schema is treated as an atomic operation, independent of how many types are modified in the process.

The type model includes the predefined types of character, integer, string, and boolean. Programmers can define new types using the following constructors: record, bounded and unbounded array, set, multi-set, union, enumeration, subrange, pointer, and alias. The type model does not include inheritance. Instead, when translating from a programming language that has inheritance to the type model, the inheritance is performed by the translator. That is, all fields that would be inherited by a subtype are included in the translated subtype's definition directly. This keeps the type model simple and allows the algorithms to cope with languages with varying subtype semantics.

Data is organized into objects. Each instantiation of a type results in the creation of a new object. Each object has an object identifier to allow objects to reference each other. Each object is tagged with its type. An object can be made persistent at any time. When an object is made persistent, all other objects reachable from that object are also made persistent. Any object can serve as the root of such a persistent structure.[2]

Because the type model does not a priori restrict persistence in any way, the schema evolution support must be very general as it must support changes to any type within a schema.

5. OBJECT CHANGES

Schema evolution is an interesting and difficult problem not just because types change, but also because the changes impact existing persistent objects. Therefore we begin by presenting a model of how objects can change as a result of schema evolution. Following that, we describe how types can change and relate type changes to object changes.

There are fundamentally three object operations associated with evolution: initialization, derivation, and deletion. New objects can be *initialized* to a default value. New objects can be *derived* from existing objects. Existing objects can be *deleted*. As derivation is the only technique that involves both existing and new objects, it is of greatest interest.

Derivation rules define how to derive new objects from existing objects. A derivation rule specifies a source type, a destination type, and a derivation function. The *source type* is a type from the schema before modification. It identifies the type of an existing object to transform. The *destination type* is a type from the schema after modification. It identifies the type of the new object to create. The *derivation function* is a function to apply to a source object to create a destination object. The simplest derivation function simply copies an existing object unmodified. A more complicated function might traverse the persistent structure starting at the source object to perform a more complex derivation such as summing a collection of values to produce a total or selecting the median from a collection of values, or it might apply a join operation to combine two related objects into one.

When evolving an object, we apply the derivation rule associated with the type of the existing object to create a new object. A derivation rule for a structured type, such as a record, is typically defined using other derivation rules. For example, to derive a new record object, it is necessary to assign a value to each new record field. The fields may be initialized to a default value or themselves derived from existing objects.

[2]This persistence model could be changed without impacting the research presented here significantly. For example, the model could allow the maintainer to restrict persistence to a subset of the types, allow only a subset of the types to be roots of persistent structures, or not automatically make all objects reachable from a persistent instance persistent.

—Local type changes:
  —Creating or deleting a type
  —Changing the name of a type
  —Changing the type constructor of a type
    —Changing an array type to a set or multi-set type, or vice versa.
    —Changing a set type to a multi-set type, or vice versa.
    —Replacing one scalar type with another.
  —Changing a type constructor argument
    —Adding an enumeration value, deleting an enumeration value, reordering enumeration values, or renaming an enumeration value.
    —Changing the lower or upper bounds of a subrange type.
    —Adding a record field, deleting a record field, reordering record fields, or changing the name of a record field.
    —Adding an array dimension, deleting an array dimension, changing the bounds of an array dimension, or reordering array dimensions.
—Reference type changes:
  —Changing the type referenced by a pointer or alias type.
  —Changing the type a subrange is defined over.
  —Changing the type of a record field.
  —Changing the index type of an array dimension.
  —Changing the type of array, set, or multi-set elements.

Fig. 1. Simple type changes.

6. SIMPLE TYPE CHANGES

We categorize simple type changes as being either local type changes or reference type changes (Figure 1 defines a complete list of all simple type changes in our type model). A *local type change* affects the structure of an individual type, such as adding a record field or changing the bounds of a subrange. A local type change affects data local to individual objects. The effects of local type changes can be expressed with derivation rules that derive each new object from a single old object. For example, a derivation rule for a record type can capture all local changes to records by initializing new fields, deleting fields no longer belonging to the record type, and providing a one-to-one mapping between the fields present in both the old and new versions of the record. The old and new fields that are paired in a record derivation rule might have different names, different types, and/or appear in a different order.

A *reference type change* replaces a type used within a type constructor with another type, such as changing the type of a record field or an array element. To fully understand how the constructed type is changed, it is necessary to understand the relationship between the old and new reference types. The effects of each reference type change are described with a derivation rule from the old reference type to the new reference type. This separation of concerns makes the derivation rules easier to understand since each derivation rule describes changes local to an individual object. For example, when deriving a new record field from an old one, we would refer to a derivation rule defined between the type of the old field and the type of the new field.

—**Inline** — Replace a type reference with its type definition.
—**Encapsulate** — Create a new type by encapsulating parts of one or more old types.
—**Merge** — Replace two or more type definitions with a new type that merges the old type definitions.
—**Move** — Move part of a type definition from one type to another existing type.
—**Duplicate** — Duplicate part of a type definition in another type definition.
—**Reverse link** — Reverse the connection between two types.
—**Link addition** — Add a link between two existing types.

Fig. 2. Compound type changes.

Since inheritance is removed during the translation from a specific programming language to the internal representation, changes to supertypes and changes to the inheritance hierarchy appear to be collections of simple type changes. For example, if a field is added to a supertype, this appears as a new field in the supertype and each subtype. If a type gains a new supertype, this appears as a collection of new fields for the type and all of its subtypes, one field for each field inherited from the supertype.

Existing database systems that support schema evolution interpret all type changes as simple type changes similar to those outlined above. Changes that make objects of a type smaller, such as deleting a record field, result in deletion of data. Those that make objects of a type larger, such as adding a record field, result in fields initialized to a default. Reference type changes result in application of the derivation rule for the reference type. The only derivation rules produced by these systems are rules that derive new objects by copying values local to the corresponding old object. Definition of nonlocal derivation rules is left to the maintainer.

7. COMPOUND TYPE CHANGES

For database systems to support nonlocal derivation rules, they must have a richer model of type changes. These *compound type changes* modify more than one type and as a result affect more than one object. Compound type changes compose three basic kinds of type operations, type deletion, type creation, and type modification, to produce more complex type changes. As with simple type changes, the fundamental *object* change that is desired is derivation of the value for a new field from the value of one or more old fields. In the case of compound type changes, however, the old and new fields belong to different types, not different versions of the same type. Each compound type change could be modeled as a collection of simple type changes, where old fields are deleted from their types and the new field is added to a different type. In doing so, however, the ability to describe nonlocal derivation is lost. In our model we include compound type changes whose effects on objects are defined with nonlocal derivation rules. In Figure 2, we list the compound type changes in our model. In this section we define the compound type changes provided by our model.

Old version:

```
type Person is
    name: string;
    address: Address;
end Person;

type Address is
    street: string;
    city: string;
    state: string;
    zipcode: integer;
end Address;
```

New version:

```
type Person is
    name: string;
    street: string;
    city: string;
    state: string;
    zipcode: integer;
end Person;
```

Fig. 3. Inlining.

7.1 Type Deletion

When deleting a type, a database maintainer is either reorganizing the type system or removing functionality. In the former case, the fields of the deleted type most likely become associated with another type, either a new type or an existing type. In these cases, the data associated with the deleted type should be moved to existing instances of the modified/created type.

One kind of compound change involving type deletion is inlining. *Inlining* involves replacing a use of a type with the type definition. Figure 3 provides an example of inlining. Here the `address` field is replaced with a collection of fields previously contained in the `Address` type. The new field values are derived from fields of the old `Address` object. If this compound type change were viewed as a collection of simple type changes, the new fields would be uninitialized and the old Address object would be deleted.

Another compound type change involving type deletion is merging. *Merging* deletes two or more types and creates a new type that represents the integration of the deleted types. Figure 4 provides an example of a merge type change. Here two or more objects must be located and combined to define a new object. In the example, `PersonalInfo` and `EmployeeInfo` objects that have the same value in their `name` field will be combined. This merge change finds its pairs of objects by joining on the `name` field. If the `name` field does not serve as a key for the two types, the results are ambiguous.

As these two examples indicate, for complex type changes to be integrated into a schema evolution system, it must be possible to identify collections of objects to modify instead of individual objects as with simple type changes. The inlining example showed a relationship between objects based on a structural connection, while the merging example showed a relationship based on equivalent values. A database maintainer could define other relationships as well.

Old version:

```
type PersonalInfo is
    name: string;
    address: Address;
    phone: Phone;
    marital_status: MaritalStatus;
    num_children: integer;
end PersonalInfo;

type EmployeeInfo is
    name: string;
    id: integer;
    salary: integer;
end EmployeeInfo;
```

New version:

```
type Person is
    name: string;
    address: Address;
    phone: Phone;
    marital_status: MaritalStatus;
    num_children: integer;
    id: integer;
    salary: integer;
end Person;
```

Fig. 4. Merge.

7.2 Type Creation

There are two type creation operations that are analogous to the type deletion operations. The merging compound type change discussed above also involves type creation. The second type creation operation is encapsulation. *Encapsulation* produces the opposite effect of inlining. Here one or more fields are replaced with a single field. The type of the new field includes the old field type(s) as reference type(s). An example of encapsulation can be seen by swapping the old and new versions in Figure 3. As with inlining, the relationship between objects is structural.

7.3 Type Modification

Compound type changes may involve the modification of types without requiring types to be created or deleted. There are four kinds of type changes fitting this description: *moving*, *duplication*, *link reversal*, and *link addition*.

Both moving and duplication involve deriving a new field from an old field. The difference is that moving deletes the original field while duplication maintains the original field. As with simple type changes, the derivation associated with moving and duplication may derive a new value, not just copy the old value. Figure 5 shows the *address* and *phone* fields being moved from the *Personal_Info* type to the *Person* type. In this case, the corresponding objects are identified using their structural relationship, specifically, the *Personal_Info* and *Person* objects that are connected using the *Person.personal* field are modified together. Figure 6 shows duplication between objects with a value relationship. Here the *id* field is duplicated from the *EmployeeInfo* object to the *PersonalInfo* object with the same value in their *name* fields.

Link reversal involves reversing the direction of a pointer. For example, consider Figure 7. Originally, the *PersonalInfo* type has a pointer to the *EmployeeInfo* type. In the modified version, *EmployeeInfo* has a pointer to

Old version:

```
type Person is
    name: string;
    personal: Personal_Info;
end Person;

type Personal_Info is
    address: Address;
    phone: Phone;
    marital_status: MaritalStatus;
    num_children: integer;
end Personal_Info;
```

New version:

```
type Person is
    name: string;
    address: Address;
    home_phone: Phone;
    personal: Personal_Info;
end Person;

type Personal_Info is
    marital_status: MaritalStatus;
    num_children: integer;
end Personal_Info;
```

Fig. 5. Moving using a structural relationship.

Old version:

```
type PersonalInfo is
    name: string;
    address: Address;
    phone: Phone;
    marital_status: MaritalStatus;
    num_children: integer;
end PersonalInfo;

type EmployeeInfo is
    name: string;
    id: integer;
    salary: integer;
end EmployeeInfo;
```

New version:

```
type PersonalInfo is
    name: string;
    id: integer;
    address: Address;
    phone: Phone;
    marital_status: MaritalStatus;
    num_children: integer;
end PersonalInfo;

type EmployeeInfo is
    name: string;
    id: integer;
    salary: integer;
end EmployeeInfo;
```

Fig. 6. Duplication based on value relationship.

the *PersonalInfo* type. Here we are reversing the structural relationship between two types.

Link addition involves adding a link between two existing types. The difference between this change and the simple type change of adding a record field is that in link addition we expect the value of the new link field to be an existing object, while when adding a record field we expect to create a new value for the new field. Once again, we can use either structural or value relationships to identify pairs of objects to add a link between. For example, Figure 8 shows the addition of an inverse link between two structurally connected types.

7.4 Limitations of the Compound Type Change Model

While this model of compound type changes allows a schema evolution system to develop nonlocal derivation rules, the maintainer still needs to be involved directly in the definition of derivation rules for two reasons. First, the default for both local and nonlocal derivation rules is to copy old values. If the maintainer wants to use a different function, such as summing a

Old version:

```
type PersonalInfo is
    name: string;
    address: Address;
    phone: Phone;
    marital_status: MaritalStatus;
    num_children: integer;
    emp_info: EmployeeInfo;
end PersonalInfo;

type EmployeeInfo is
    id: integer;
    salary: integer;
end EmployeeInfo;
```

New version:

```
type PersonalInfo is
    address: Address;
    phone: Phone;
    marital_status: MaritalStatus;
    num_children: integer;
end PersonalInfo;

type EmployeeInfo is
    name: string;
    id: integer;
    salary: integer;
    private_info: PersonalInfo;
end EmployeeInfo;
```

Fig. 7. Link reversal.

Old version:

```
type PersonalInfo is
    name: string;
    address: Address;
    phone: Phone;
    marital_status: MaritalStatus;
    num_children: integer;
    emp_info: EmployeeInfo;
end PersonalInfo;

type EmployeeInfo is
    id: integer;
    salary: integer;
end EmployeeInfo;
```

New version:

```
type PersonalInfo is
    name: string;
    address: Address;
    phone: Phone;
    marital_status: MaritalStatus;
    num_children: integer;
    emp_info: EmployeeInfo;
end PersonalInfo;

type EmployeeInfo is
    id: integer;
    salary: integer;
    private_info: PersonalInfo;
end EmployeeInfo;
```

Fig. 8. Link addition.

collection of values or finding a median, the maintainer must provide this function explicitly. Second, the merge, move, duplicate, and link addition type changes require finding collections of old objects to operate on. It may be necessary for the maintainer to indicate how to find matching objects to operate on, particularly if the relationships are not structural or by equivalent values.

8. EXAMPLE

Compound type changes can be combined to produce interesting schema changes whose effects on existing data can be understood following the model given. In this section, we provide an example of a real schema evolution and describe how it fits into the compound type change model.

Figure 9 shows consecutive versions of a collection of interrelated types extracted from TAOS, a software testing tool [Richardson 1993]. In this example, we see three modified types and four new types.

Old version:

```
type SaveTestCases is (nada, todo);

type RandomTestInfo is
    MinLength: natural := 0;
    MaxLength: natural := 0;
    NumberRequired: positive := 1;
    Persistence: SaveTestCases := todo;
    NumberNonPersistentPassed:
        natural := 0;
    NumberNonPersistentFailed:
        natural := 0;
end;

type TestClass is
    ExtraInfo: RandomTestInfo;
end ;
```

New version:

```
type TestCaseState is
    (Pass, Fail, Untested);

type SaveTestCases
    is array ( TestCaseState ) of boolean;

type RandomTestInfo is
    MinLength: natural := 0;
    MaxLength: natural := 0;
    NumberRequired: positive := 1;
end;

type Saved is ( persistent, nonpersistent );

type TestCaseCounts is
    array ( Saved, TestCaseState )
        of natural;

type TestCasesInfo is
    PersistencePreferences: SaveTestCases
        := Default_Persistence;
    NumTestCases: TestCaseCounts :=
        Default_Counts;
end;

type TestClass is
    TestSetInfo : TestCasesInfo := Create;
    ExtraInfo: RandomTestInfo;
end;
```

Fig. 9. Schema evolution in TAOS.

If we consider each type in isolation, we see the following simple type changes: three fields have been deleted from `RandomTestInfo`, one field has been added to `TestClass`, `SaveTestCases` has changed from an enumerated type to an array of booleans, and four new types have been created. Treating these as simple type changes would result in the deletion of the values associated with the deleted fields of `RandomTestInfo`, the initialization of the new field in `TestClass` to its default value, and the deletion of values of the `SaveTestCases` type.

Now, let's reconsider the example as a sequence of compound type changes.

—Moving: The `Persistence`, `NumberNonPersistentFailed`, and `NumberNonPersistentPassed` fields are moved to the `TestClass` type using a structural relationship.

```
type RandomTestInfo is
   MinLength: natural := 0;
   MaxLength: natural := 0;
   NumberRequired: positive := 1;
end;
type TestClass is
   Persistence: SaveTestCases := todo;
   NumberNonPersistentPassed: natural := 0;
   NumberNonPersistentFailed: natural := 0;
   ExtraInfo: RandomTestInfo;
end;
```

—Encapsulation: The `NumberNonPersistentFailed` and `NumberNonPersistentPassed` fields are encapsulated into a new field named `NumTestCases` whose type is the new `TestCaseCounts` type. Specifically, the value of the `NumberNonPersistentFailed` field is moved to the `TestCaseCounts` element indexed by `(nonpersistent, Fail)`. The value of the `NumberNonPersistentPassed` field is moved to the `TestCaseCounts` element indexed by `(nonpersistent, Pass)`.

```
type TestCaseState is (Pass, Fail, Untested);
type Saved is (persistent, nonpersistent);
type TestCaseCounts is array (Saved, TestCaseState) of natural;
type TestClass is
   Persistence: SaveTestCases:= todo;
   NumTestCases: TestCaseCounts:= Default_Counts;
   ExtraInfo: RandomTestInfo;
end;
```

—Encapsulation: The `Persistence` field is encapsulated into an attribute named `PersistencePreferences` whose type is the new `SaveTestCases`. The value of the field is duplicated in each element of the `SaveTestCases` array, translating `nada` to `false` and `todo` to `true`.

```
type TestCaseState is (Pass, Fail, Untested);
type SaveTestCases is array (TestCaseState ) of boolean;
type TestClass is
   PersistencePreferences: SaveTestCases:= Default_Persistence;
   NumTestCases: TestCaseCounts:= Default_Counts;
   ExtraInfo: RandomTestInfo;
end;
```

—Encapsulation: `PersistencePreferences` and `NumTestCases` are encapsulated into a new field named `TestSetInfo` whose type is the new `TestCasesInfo` type.

```
type TestCasesInfo is
   PersistencePreferences: SaveTestCases:= Default_Persistence;
   NumTestCases: TestCaseCounts:= Default_Counts;
end;
type TestClass is
   TestSetInfo: TestCasesInfo:= Create;
   ExtraInfo: RandomTestInfo;
end;
```

This example demonstrates the type change model. It also demonstrates that describing these type changes via editing commands would be cumbersome. We have developed type comparison algorithms to support such changes without requiring the user to specify them explicitly.

9. TYPE COMPARISON

For our type comparison approach to be feasible, we assume that between successive versions of a system, most type definitions remain mostly the same. We rely heavily upon the similarities that exist to quickly prune the space of types that must be compared. Since the types in databases tend to experience evolutionary change, rather than revolutionary change, we do not expect this to be a significant problem for most situations. In those situations in which revolutionary changes occur, the maintainer can and should provide more guidance rather than relying on the fully automated control algorithm. In Section 10.1, we describe how the maintainer can provide guidance in our implementation of the type comparison algorithms.

In this section, we describe derivation rules in more detail. Next, we describe the algorithm that controls which types are compared. Following that, we describe the algorithm to recognize simple changes that may occur in a record definition. Then, we present the algorithm to identify movement of fields between structurally-connected records, including encapsulation and inlining. Finally, we explain how derivation rules could be used with a variety of data translation mechanisms.

9.1 Derivation Rules

A *derivation rule* describes how to translate data created using one type definition to a different type definition. For simple values, such as integers and enumerated types, the derivation rule defines a function to apply to the old value to compute the new value. In the simplest derivation rules, the function is simply an identity function. For example, suppose we have a *Counter* type in our old and new schema. Assume that this *Counter* type is unmodified. The corresponding derivation rule is the following:

```
Counter ⇒ Counter:
  new:= old;
```

Note the use of the keywords *old* and *new*. *old* refers to the existing data that we are translating from. *new* refers to the new data that we are creating. In this case, the new data has the same value as the old data.

For structured types, such as records, the function specifies how to compute the value for each new substructure of the new type. Usually, the value of a new substructure is defined in terms of the value of existing data. As a result, the derivation of most substructures is performed by applying the derivation rule defined between the types of the corresponding substructures. A new substructure may be defined using a constant value or a user-supplied function. For example, Figure 10 is the derivation rule that corresponds to Figure 5.

```
Person ⇒ Person:
    new.name : derive from old.name;
    new.address : derive from old.home.address;
    new.home_phone : derive from old.home.phone;
    new.home : derive from old.home;
```

Fig. 10. Derivation rule for a structured type.

In this case, the new fields are all derived from existing data. For example, the value for the new *personal* field is computed by applying the derivation rule from the old *Personal_Info* type to the new *Personal_Info* type.

In some cases, we may want to use slightly different derivation rules between a pair of types depending upon the state of the database. For example, suppose we replace one type with a collection of types. The intent may be to partition the existing values so that each value belongs to one of the new types. To support this, we add conditionals to our derivation rules. Consider the type change and corresponding derivation rule shown in Figure 11. Here we will create a different type of new object depending on the value of an existing field.

A *similarity metric* is associated with each derivation rule. A similarity metric is a qualitative description of the impact that applying the derivation rule would have on existing persistent data. For example, derivation rules between record types have one of the following similarity metrics:

Each derivation rule has a single metric that describes the worst effect of applying the rule. Thus a rule with a *NewField* metric may also have fields whose names have changed, but it will not have any deleted fields. Similarity metrics are used within the comparison algorithms to prune the space of comparisons considered. (In Section 10.1, we will also describe how similarity metrics are used to focus the maintainer's attention on the derivation rules with greatest impact on the data.)

9.2 The Type Comparison Control Algorithm

The input to the type comparison algorithms is the set of type definitions of consecutive schemas. The algorithms selectively compare the types to identify how the types have changed and output derivation rules describing how to transform instances of the old version into instances of the new version. The type comparison control algorithm is responsible for determining which types to compare, based primarily on the results of comparisons done thus far and on naming similarities between old and new types, as well as which comparison algorithms to use based on the type constructors used by the types being compared. The algorithms ignore changes to white space and comments and the order in which the type definitions appear in the schema.

The fully-automated type comparison control algorithm is shown in Figure 12. It proceeds through three stages. First, in the *name comparison* stage, old and new types that have the same names in both versions are

Old version:

```
type Plane is
    engine: EngineType;
    num_passengers: positive;
    max_speed: positive;
end;

type PlaneFleet is set (Plane);
```

New version:

```
type Jet is
    num_passengers: positive;
    max_speed: positive;
end;

type PropellorPlane is
    num_passengers: positive;
    max_speed: positive;
end;

type Glider is
    num_passengers: positive;
    max_speed: positive;
end;

type Plane is union of (Jet, PropellorPlane, Glider);

type PlaneFleet is set (Plane);
```

```
PlaneFleet ⇒ PlaneFleet
    for each old_plane in old
        if old_plane.engine == JetEngine
            new_plane = new Jet derived from old_plane
        else if old_plane.engine == PropellorEngine
            new_plane = new PropellorPlane derived from old_plane
        else if old_plane.engine == None
            new_plane = new Glider derived from old_plane
        end if;
        insert new_plane in new
    end for;
```

Fig. 11. Conditionals in derivation rules.

Table I.

Similarity Metric	Meaning
Identical	No changes to the type
FieldOrderChange	Fields appear in a different order.
FieldTypeNameChange	Name of the type of a field has changed.
FieldNameChange	Name of a field has changed.
NewField	New type has an extra field.
DeletedField	Old type has an extra field.

compared. For structured types, such as records and arrays, this may result in further type comparisons. For example, a derivation rule that derives a new array from an old array requires a derivation rule from the old array element type to the new array element type. Comparing these element types is called *component comparison*. In the second stage, called *use site comparison*, types that use types that have been successfully compared are compared. In the final stage, called *exhaustive comparison*, each old type that does not already have a derivation rule is compared to each new type, first considering only those new types that use the same

```
procedure CompareTypes (old_types, new_types) is
begin
    - Compare types with the same name.
    for  each type o in old_types
        let n = type in new_types with the same name as o
        if Compare (o, n) finds a derivation rule then
            add (o, n) to TypePairList
        end if;
    end for ;

    - Check the use sites for each pair of types that have a derivation rule.
    for  each type pair tp in TypePairList
        let o = old type in tp
        let n = new type in tp

        - Find where the old and new type are used.
        let old_uses = set of types in old_types that use o
        let new_uses = set of types in new_types that use n

        - Compare each pair of use sites.
        for  each type o_u in old_uses
            for  each type n_u in new_uses
                if Compare (o_u, n_u) finds a derivation rule then
                    add (o_u, n_u) to TypePairList
                end if;
            end for ;
        end for ;
    end for ;

    - Exhaustive search
    for  each type o in old_types
        - Make sure we have at least one derivation rule for each old type
        if there is no derivation rule from o to any type in new_types then
            - Compare to new types with the same type constructor.
            for  each type n in new_types with the same type constructor
                if Compare (o, n) finds a derivation rule above threshold then
                    compare the use sites of o and n; break;
                end if;
            end for ;

            - Compare to new types with different type constructors if no good match to the same type constructor.
            if there is no derivation rule from o to any type in new_types then
                for  each type n in new_types with a different type constructor
                    if Compare (o, n) finds a derivation rule above threshold then
                        compare the use sites of o and n; break;
                    end if;
                end for ;
            end if;
        end if;
    end for ;
end;
```

Fig. 12. Type comparison control algorithm.

type constructor and, if that fails to produce an acceptable derivation rule, considers all remaining new types. The exhaustive comparison algorithm also performs component comparisons and use site comparisons as derivation rules are generated. Thus if a derivation rule is found by exhaustive comparison, the algorithms immediately compare pairs of types used by the matched type pair as well as pairs of types using the matched type pair. This further reduces the search for matching types. Since we do not compare each pair of types, it is possible that we will miss the correct mapping. In practice, we have found this to not be a problem, but rather

```
type old_record is record
    field1: old_field1_type;
    field2: field2_type;
end record;
```

```
type new_record is record
    field1: new_field1_type;
    field2: field2_type;
end record;
```

Fig. 13. Component and use site comparisons.

have found that examining type names and structural relationships is generally sufficient to find the changes we have encountered in real systems. We will discuss this further in Section 11.

To better understand the stages of the algorithms, consider the type definitions in Figure 13. *old_record* is an old type and *new_record* is the corresponding new type. Since they have different type names they are not compared during the name comparison phase. The two versions of *field2_type* (not shown) are compared in this phase. Assuming a derivation rule is found between these types, the use site comparison stage searches for pairs of types that use *field2_type*. It finds *old_record* and *new_record* and compares them. To complete the comparison of *old_record* and *new_record*, *old_field1_type* and *new_field1_type* are compared during component comparison since they have the same field name.

9.3 Recognizing Simple Type Changes: A Sample Algorithm

When looking for simple changes between two types, the algorithm varies depending upon the type constructors that the types use. For example, a different algorithm is used to compare two enumerated types than to compare two record types. We have also developed algorithms to compare two types that use different type constructors, such as sets and arrays.

In Figure 14 we show the algorithm that compares two records to give more insight into how the type comparisons proceed. The input to this algorithm is the type definitions of two record types. The output is a derivation rule between those record types, such that each record field of the new type is either derived from an old record field or is initialized to a default value, and each record field of the old type that is not used in a derivation is explicitly identified as being deleted.

The record comparison algorithm is quite similar to the algorithm used to compare the sets of type definitions. First, it compares record fields with the same name. Next, it compares old unmatched fields with new unmatched fields with the same type name. If there are multiple old and new fields with the same type, they will be paired in the order in which they occur. We could produce multiple mappings, one for each pair of fields with the same type to account for changes where the name and ordering of the fields have been changed simultaneously, but this has not been a problem in practice. Finally, the algorithm compares each old unmatched field to each new unmatched field. Again, we could produce a mapping for each permutation of old unmatched fields to new unmatched fields to account for simultaneous name, order, and type changes to fields. In practice, however, this is likely to lead only to spurious results. Based on our experience, it

```
function CompareRecords (old_record, new_record) return derivation_rule is begin
    let r = new derivation rule from old_record to new_record;

    - Compare fields with the same name
    for each field o_f in old_record
        for each field n_f in new_record
            if o_f and n_f have the same names
                if Compare (type of o_f, type of n_f) finds a derivation rule above threshold
                    map o_f to n_f in r;
                end if;
                break;
            end if;
        end for;
    end for;

    - Compare fields with the same type name
    for each field o_f in old_record
        if o_f is not used
            for each field n_f in new_record
                if n_f is not mapped to any new field and o_f and n_f have different names and
                    o_f and n_f have the same type names then
                    if Compare (type of o_f, type of n_f) finds a derivation rule above threshold
                        map o_f to n_f in r;
                    end if;
                    break;
                end if;
            end for;
        end if;
    end for;

    - Compare unmatched old fields to unmatched new fields
    for each field o_f in old_record
        if o_f is not used
            for each field n_f in new_record
                if n_f is not mapped to any new field and o_f and n_f have different names and
                    o_f and n_f have different type names then
                    if Compare (type of o_f, type of n_f) finds a derivation rule above threshold
                        map o_f to n_f in r; break;
                    end if;
                end if;
            end for;
            if o_f is not mapped to any new field then
                mark o_f as deleted; end if;
            end if;
        end for;

        for each field n_f in new_record
            if no old field is mapped to n_f then
                mark n_f as initialized to default;
            end if;
        end for;
    end;
```

Fig. 14. Record comparison algorithm.

seems more likely to have a field deletion and addition than a derivation if

neither the field name, type, nor ordering is preserved. When the algorithm compares fields, it compares the field names and recursively compares the field types. If type definitions are recursive, as with linked lists for example, recursive comparison of field types leads to an infinite loop. To avoid this, we use an algorithm similar to the one used by Amadio and Cardelli to check subtyping of recursive types [Amadio and Cardelli 1993], which limits the recursion performed when comparing recursive types. We cache the results of type comparisons in a matrix so that we can look up the results of previous comparisons instead of repeating them.

Upon completion of the record comparison algorithm, any field of the old type that has not been mapped to a new field represents a field whose value will be deleted during transformation. Any field of the new type that does not have an old field mapped to it represents a field whose value will be initialized to a default value during transformation.

Using algorithms such as the one described here we can recognize changes equivalent to those supported by databases that provide automatic support for schema evolution, including Orion and GemStone.

9.4 Recognizing Compound Type Changes: A Sample Algorithm

We have also developed algorithms to recognize compound changes. This allows us to support type changes not supported by other databases. Figure 15 shows the algorithm for recognizing movement and encapsulation of fields from one record type to another where the types are structurally related. This algorithm is passed an old record type, a new record type, and the derivation rule constructed by the algorithm to detect simple changes in record types. In this initial derivation rule, the old fields that are marked as deleted might become sources of movement while the new fields marked as initialized to default values might become destinations of movement. After applying this algorithm, the derivation rule identifies the structural connections that must be traversed to move the data from those old fields that would be deleted to new fields that would be initialized.

To accomplish this task, the algorithm identifies all the unused fields of the old type in the derivation rule. It also transitively finds all unused subfields of any field of the old type. These are fields that might be moved. It then constructs a dummy record definition whose fields are these unused fields and subfields. The field names used in these dummy record definitions encode the path to the real subfield so that this information can be used to identify the source of a moved field. In a similar manner, a second dummy record definition is created to hold all the unused fields and subfields of the new type, again encoding the path to the real subfield. Next, the algorithm applies the record comparison algorithm for recognizing simple type changes given in Figure 14. Field mappings identified by comparing these two dummy types necessarily involve a subfield from the old, new, or perhaps both types, since all mappings between fields of the old and new type have been identified prior to calling the compound type comparison algorithm. These mappings correspond to compound type

```
procedure RecordFieldMove (old_record, new_record, deriv_rule) is
begin
    create an empty record type o
    for each field o_f in old_record
        if o_f is marked as deleted in deriv_rule then
            add o_f to o;
        end if;
        add deleted fields and subfields of o_f in deriv_rule to o
    end for

    create an empty record type n
    for each field n_f in new_record
        if n_f is initialized in deriv_rule then
                add n_f to n;
        end if;
        add initialized fields and subfields of n_f in deriv_rule to n
    end for

    move_rule = CompareRecords (o, n);
    for each field mapping fm in move_rule
        copy fm to deriv_rule
    end for;
end;
```

Fig. 15. Record field move algorithm.

changes. Each field mapping identified by this algorithm is then merged into the original derivation rule. In this way, the derivation rule now encodes both local and nonlocal changes.

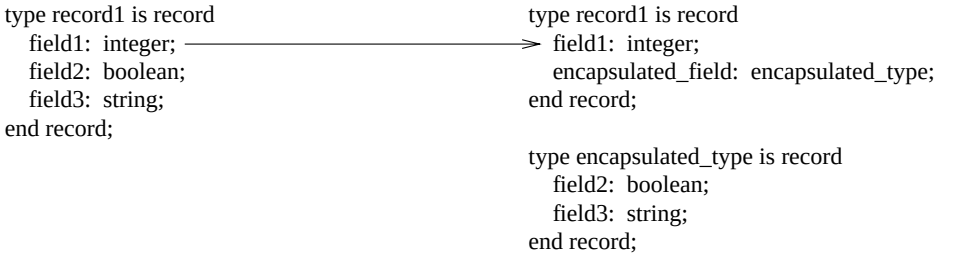
Figure 16 graphically shows the derivation rules that the encapsulation algorithm finds for a particular set of type definitions. The top of the figure shows the result of comparing the old and new versions of `record1` looking for only simple type changes. The `field2` and `field3` fields of the old version are unused; the `encapsulated_field` of the new version is unused. After looking for compound type changes, `field2` is mapped to `encapsulated_field.field2` and `field3` is mapped to `encapsulated_field.field3`.

Using algorithms such as this one we can recognize compound type changes where the relationships between the source and destination types are structural.

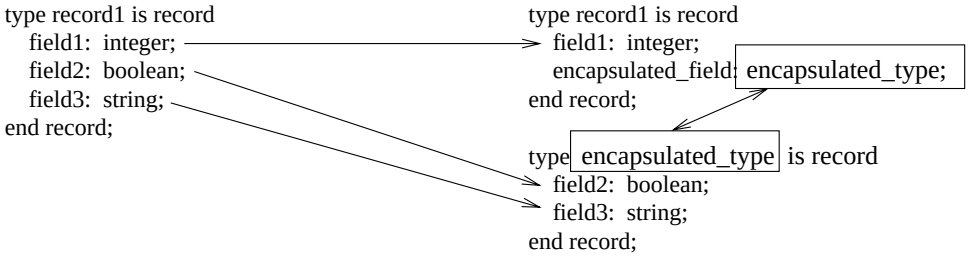
9.5 Using Derivation Rules to Perform Data Translation

The derivation rules do not specify when objects are updated or whether those updates are persistent. This separation is done deliberately to allow derivation rules to be used by a variety of data translation mechanisms. In this section, we briefly describe how the derivation rules would be used by several different data translation mechanisms.

9.5.1 *Conversion*. In a conversion backend, data are read from their old format, converted to the new format, and written back to the database. GemStone and O_2 are examples of object-oriented database systems that



Before Checking for Encapsulation



After Checking for Encapsulation

Fig. 16. Encapsulation example.

perform conversion. Conversions can be performed by taking the database offline, starting at each root object, visiting each object in turn, converting it, and writing the result to the database.

If the database is large or high availability is required, it may not be feasible to take the database offline. In these cases, lazy conversion can be done as in O_2 [Ferrandina et al. 1994]. With lazy conversion individual objects are converted as they are accessed. To work with conversion, we would apply a derivation rule locally, but would only convert components of structured objects as they were accessed. Ferrandina presents a solution to the problem of ordering conversions in lazy conversion to ensure that data is not deleted before it is transformed. Our derivation rules can be used with his algorithms.

9.5.2 *Screening*. With screening, information is never deleted from objects. Instead, the accessing functions hide the appropriate information based upon the version of the code that is accessing the object. Orion [Banerjee et al. 1987; Kim and Korth 1988] uses screening to evolve its objects. To do so, it uses a clever object representation that allows fast access to objects even after the object's type has been changed. It also restricts the kinds of changes to a subset of our local change model. The changes that they allow minimize the impact on the persistent objects, but reduce the flexibility available to the maintainer.

Derivation rules can extend the screening approach to more complex type changes. This is accomplished by applying the derivation rules as objects are accessed. For example, suppose we want to move data from one object to another conceptually. When an object is accessed from which data has moved, the data should be hidden by the accessing function, just as deletion is managed currently. When an object is accessed to which data has moved, the accessing function must apply the portion of the derivation rule that defines where the data comes from to access the data. The advantage of this approach is that it has minimal impact on the data, just as current screening techniques. Furthermore, there is no need to take the database offline. The disadvantage is that accessing objects associated with these complex type change operations will pay a penalty on each access. Additionally, we must be careful to not delete objects that contain data that serves as the source of a data movement operation.

9.5.3 *Versioning*. In a versioning backend, such as Encore [Skarra and Zdonik 1986] and O_2 provide, multiple versions of an object may exist at one time. The runtime system compares the version of code accessing an object with the versions available so that the correct version can be returned. If the correct version does not exist, it is created dynamically by applying the appropriate derivation rules. Current systems that use versioning only support local type changes. In order to derive a new version of an object, one only needs to access an existing version of that object. Our derivation rules could be used in a similar manner to support nonlocal derivations.

To support this, derivation rules must be able to translate from newer versions of a type system to older versions. This could be done by applying the same comparison algorithms but changing which is being used as the source and which as the destination. A more straightforward technique would be to define reverse transformations directly by analyzing the existing derivation rules.

Thus, we see that the basic notion of derivation rules is quite flexible and could be used with a variety of data translation mechanisms to address different database concerns such as minimizing access times, maintaining high availability, and reducing the impact on existing code.

10. TESS: AN EXPERIMENTAL TYPE COMPARISON SYSTEM

We have implemented type comparison algorithms in a tool called Tess. Tess can automatically generate derivation rules for all simple type changes listed in Figure 1. It can also generate derivation rules for inlining, encapsulation, link reversal, link addition, merging of structurally-connected types, and moving fields between structurally-connected types. The maintainer has extensive control over what is automated: Tess can operate in modes ranging from completely manual, as in early database systems, to fully automated. The combination of powerful derivation rule generation algorithms and flexible application of those algorithms as determined by a maintainer leads to a synergism not found in existing systems. In this

section, we describe the maintainer's role in running Tess, how Tess assures that all necessary derivation rules have been provided, and then discuss experimental results.

10.1 Maintainer Control over Type Comparison

Tess has an interactive user interface that gives the maintainer control over how type comparisons proceed. There are three dimensions that the maintainer has control over. First, the maintainer can control which stages of the comparison control algorithm are used (name comparison, use site comparison, and exhaustive search).

The second dimension that the maintainer can control is which types get compared. Here there are three options. All types can be considered at once (the fully-automated control algorithm shown in Figure 12), a specific old and new type can be compared, or an individual old type can be compared with all new types. When schema changes are minor, it is reasonable for the maintainer to use all stages of type comparison and allow all types to be compared. When the schema has changed radically, it would be better to use only the name comparison algorithm on all types to identify the unchanged types and obvious changes and then complete the transformer by specifying pairs of types for Tess to develop derivation rules for.

The third dimension involves determining which derivation rules are automatically accepted as correct and which must be presented to the maintainer for manual acceptance. This is done by defining a threshold value for the similarity metric. The most conservative approach accepts only derivation rules between unchanged types, that is, simple identity rules. A more liberal policy accepts changes in which all old values still belong to the new type, such as increasing the size of a subrange. The most liberal, yet still sensible, policy automatically accepts those changes that affect the representation of the type within the database, but not its use within a program, such as reordering the fields of a record. If a schema contains many unchanged types or trivial changes, the use of similarity metrics allows the user to quickly focus on the interesting changes.

10.2 Assuring Completeness of Derivation Rules

Since generation of derivation rules is a separate activity from updating the persistent data, it is important that all the necessary derivation rules are produced so that they are available when we later attempt to access old data. In this section, we describe how we ensure this.

Recall that our model of persistence assumes that any type can be the root of a persistent structure. As a result, we require a *root derivation rule* for each old type. The root derivation rule is used to translate an instance of a type that appears as an old root into a new root defined using the new schema. Frequently, the old and new types of a root derivation rule have the same type name, but this is not necessarily so.

Recall also that when we apply a derivation rule that creates a structured type, it generally assigns values to the components of the structure

Old version:

```
type Person is
    name: string;
    age: integer;
end Person;
```

New version:

```
type Person is
    name: string;
    birthday: Date;
end Person;
```

Fig. 17. Determining which derivation rules are required.

by applying another derivation rule (as in Figure 10). Unlike root derivation rules, Tess can determine which old and new types the derivation rule must operate on by examining the types of the components that are paired. These derivation rules are referred to as *reference derivation rules*. For each accepted derivation rule (root or reference), we examine the derivation rules used within the accepted rule to determine which pairs of types require reference derivation rules.

For example, consider Figure 17. When Tess identifies the mapping between the two versions of the *Person* type, it identifies this as a root mapping. As a result, if it encounters an object of type *Person* that is not referenced by any other object, it will transform it to a new *Person* object. The transformation function would translate the old *name* field to the new *name* field. It therefore requires a transformation from *string* to *string*. Similarly, to transform the old *age* field to the new *birthday* field, it needs a transformation function from the *integer* type to the new *Date* type.[3] In this case, we do not want to transform all integers in the program to dates, but only those integers used within *Person* objects. Thus, the derivation rule from *integer* to *Date* is a reference derivation rule. It is quite common for the same rule to serve as both a root and reference derivation rule.

Using this information, Tess keeps track of which types still require derivation rules. A user may decide that not all types actually are used as the roots of structures and thus the user can tell Tess that root derivation rules are not required for those types. In contrast, the analysis of which reference derivation rules are required is precise. If a reference derivation rule is missing, a runtime error would occur if the derivation rule that used the missing reference derivation rule was applied. Tess displays this status information to the user, indicating which old types do not yet have root derivation rules and which type pairs referenced by accepted derivation rules do not have derivation rules. Requiring completeness ensures that we will be able to find the appropriate transformation function for any old data that we might encounter.

11. EVALUATION

The algorithms within Tess generally use three mechanisms to create derivation rules. First, they use naming similarities. Given the significance

[3]Obviously, this transformation function must be written by a programmer and even then can only approximate the actual birthdate.

of names to humans, it seems reasonable to expect that derivation rules that rely on naming similarities are likely to be correct.

Second, they use structural information. Specifically, they examine the type constructors used by the old and new types and the types of components. The former allows Tess to identify situations in which a unique value is replaced with an array of values of the same type for instance. The latter is particularly important when the component names have been changed but their types are unchanged. Given the significance of types for programmers, we expect these algorithms to yield good results, particularly in cases where the types involved are user-defined types (not integer, for example) and where there is only one component using that type in the old and new types so that ambiguity does not arise.

Third, when all else fails, the algorithms rely upon ordering information. For example, Tess will attempt to map components with different names and different types if it has no other alternative. We expect ordering to be the least useful of the heuristics used.

Currently, the similarity metrics represent the nature of the change that has been identified but not the nature of the algorithm used (naming, component typing, or ordering). This ensures that the user will evaluate derivation rules based upon their effect on the data rather than upon the way in which the derivation rule was created. As we evaluate Tess's performance in this section, however, it is useful to consider the nature of the algorithms involved in considering their effectiveness. In the remainder of this section, we present a detailed case study of the use of Tess on a real application and summarize other experimentation.

11.1 Case Study

Figure 18 shows the results of applying Tess to the example shown in Figure 9. The compound change from the old *TestClass* type to the new *TestClass* type is correctly identified. If the compound change algorithm had not been applied, the *TestClass* to *TestClass* derivation rule would have initialized the *TestSetInfo* field to a default value. With the compound change algorithm, we see that the *Persistence*, *NumberNonPersistentPassed*, and *NumberNonPersistentFailed* fields are moved from the *RandomTestInfo* type to the *TestCasesInfo* type. Objects that the data should move between are connected structurally through the old and new *TestClass* type. The data moves from the old *TestClass.ExtraInfo* to the new *TestClass.TestSetInfo* field. Of particular interest is the movement of data from *TestClass.ExtraInfo.Persistence* to *TestClass.TestSetInfo.PersistencePreferences*. The movement is accomplished by applying the reference derivation rule between the old and new *SaveTestCases* types. The definition of the *SaveTestCases* type has changed considerably, however. In the old version, it was an enumerated type of two values. In the new version, it is an array of booleans. The derivation rule generated by Tess specifies that the old value should be placed in the first element of the new array, applying the derivation rule between *SaveTestCases* and *boolean* to compute the new value.

This is an interesting case study for several reasons. First, it demonstrates Tess's ability to recognize complex type changes. Second, it demonstrates a diverse set of algorithms used to recognize those changes. Third, its failings demonstrate the need for continued human involvement in the development of powerful derivation rules.

The algorithm to map the old *RandomTestInfo* type to the new *RandomTestInfo* type relies solely upon matching the type names and the names of the components within the types. As a result, we expect the rule to be correct. Since it results in values being deleted, Tess attempts to find a complex type change that can account for the deleted components. It does so in the derivation rule from the old *TestClass* type to the new *TestClass* type. Here the types have the same names, whose single old component can be paired with a new component based upon its name. In this case, however, we find an additional component, *TestSetInfo*, that has no old value mapped to it. The complex type change algorithm identifies the need to move data, this time, by using similarities between the component types, rather than the component names, so we have less confidence of this being correct.

The mappings from *ExtraInfo.NumberNonPersistentPassed* to *TestSetInfo.NumTestCases(persistent, Pass)*, from *ExtraInfo.NumberNonPersistentFailed* to *TestSetInfo.NumTestCases(persistent, Fail)*, between the old and new *SaveTestCases* type and between the values of the old *SaveTestCasesType* and *boolean* are based entirely upon ordering information. As a result, we have less confidence in the correctness of these changes and, indeed, these are where the errors occur in the automatically derived derivation rule. Tess did not generate the correct rule between the old and new versions of the *SaveTestCases* type. The generated rule takes the old value and places it in the first element of the new array. The correct derivation rule would duplicate the old *SaveTestCases* value into all elements of the new *SaveTestCases* array. It is not possible to distinguish between the case of insertion into a single array element and duplication in more than one array element by looking at the type definitions alone. It requires a more semantic understanding of the change and thus we expect the maintainer to provide this information. Similarly, the mapping from *ExtraInfo.NumberNonPersistentPassed* and *ExtraInfo.NumberNonPersistentFailed* go to incorrect elements of the array. Mapping into the array elements was based upon typing information and this decision was correct. The wrong array elements were chosen, however, based purely on ordering information. This case study brings into question the value of using ordering information. A better solution might be for Tess to provide partial derivation rules in these cases and explicitly require the programmer to disambiguate them. For example, Tess might propose mapping the old *SaveTestCases* type into an element of the array, but require the programmer to indicate which element(s) explicitly instead of proposing one based on ordering information.

The similarity metrics on the derivation rules shown result in Tess requiring approval of the derivation rules before they would be applied (for any reasonable threshold for automatic acceptance). Out of a total of 37

```
TestClass ⇒ TestClass: Compound Change
    new.ExtraInfo : derive from old.ExtraInfo;
    new.TestSetInfo.PersistencePreferences: derive from old.ExtraInfo.Persistence;
    new.TestSetInfo.NumTestCases(persistent, Pass) : derive from old.ExtraInfo.NumberNonPersistentPassed;
    new.TestSetInfo.NumTestCases(persistent, Fail) : derive from old.ExtraInfo.NumberNonPersistentFailed;

RandomTestInfo ⇒ RandomTestInfo: Deletes Old Component
    new.MinLength : derive from old.MinLength;
    new.MaxLength : derive from old.MaxLength;
    new.NumberRequired : derive from old.NumberRequired;

SaveTestCases ⇒ SaveTestCases: Requires New Component
    new element indexed by Pass: derive from old;
    new element indexed by Fail: uninitialized;
    new element indexed by Untested: uninitialized;

SaveTestCases ⇒ boolean: Value Change
    if old == nada
        new = false;
    else if old == todo
        new = true;
    end if;
```

Fig. 18. The derivation rules generated by Tess for the TAOS example.

derivation rules generated by the complete example, only two other derivation rules required review and both of those derivation rules were correct. Thus, even though the totally-automated algorithm did not produce a completely correct set of derivation rules, it did focus the maintainer's attention on the few complicated situations that existed. Even in the case where the derivation rule was wrong, the changes required to correct the derivation rule were quite minor relative to the overall complexity of the derivation rules.

This example demonstrates capabilities for which existing evolution systems provide no automated support. The change of *SaveTestCases* from an enumerated type to an array of booleans cannot be done in existing automated systems. The movement of fields from *RandomTestInfo* to the *TestSetInfo* field of the *TestClass* type would result in deletion of the associated data with existing automated evolution systems. Systems that require the maintainer to provide the transformation routines would allow proper handling of these transformations, but development of those routines would be entirely manual.

11.2 Experimentation

We have performed two experiments using Tess. The first experiment involved extracting old versions of systems that had been saved in a version control system and using Tess to compare the types in those versions. The second experiment involved creating examples based upon reengineering code using design patterns [Gamma et al. 1995]. In this section, we describe those experiments and the results found.

11.2.1 *Comparing Histories of Implemented Systems.* In the first experiment, we ran Tess on consecutive versions of the histories of 20 programs

Table II. Results of Experimentation with Implemented Systems

Type of change	Number of occurrences
Compound change	2
Deleted type	3
Deleted record field	13
Field type change	7
New record field	19
Field order change	1
Record field name change	3
Field type name change	23
New union member	1

developed over a period of 5 years. Most of these programs were created within the Laboratory for Advanced Software Engineering Research at the University of Massachusetts, Amherst. The most versions there were of a single program were 5. While it is difficult to know if these program histories are representative of a larger population, it is the case that many of the histories were created before Tess was started and all were created before Tess was ready for use. Therefore, the types of changes made were not influenced by the types of changes that Tess could handle.

There were two goals to this experiment. The first was to determine the kinds of type changes that are likely to occur in practice. The second was to determine whether or not Tess could handle those changes. Table II summarizes the type changes contained in these files as analyzed by a human.[4] Of these, Tess correctly identified all the type changes except for the type deletions. Tess is programmed to attempt to identify derivation rules for all types. For each deleted type, instead of indicating that the type was deleted, it proposed several different mappings that had low similarity metrics. Since these all had low similarity metrics, the user would be expected to review the rules and could very easily delete them.

Most type changes encountered in this experiment are simple ones. There were only two compound type changes. One was described in detail in the case study in Section 11.1. The second was an encapsulation in which a field that contained a single instance of a type was replaced with a type that could hold a set of the original type. Tess created correct derivation rules for all changes.

11.2.2 *Comparing Types Reengineered with Design Patterns*. The first experiment did not exercise the compound type change algorithms much. To better assess the effectiveness of the compound type change algorithms, we devised examples based upon the structural design patterns in Gamma et al. [1995]. For each pattern, we first devised an example that did not use the pattern. We then reengineered the example using the pattern. In this

[4]Six of the field type changes were from integer to unsigned integer; one was from unsigned integer to string. The field type name changes were all situations in which the name of an externally-defined type was changed, but its definition was not.

way, we were evaluating Tess on examples derived from the literature and also representative of the kinds of type changes that we might expect to encounter as programmers incorporate patterns into existing code. The design patterns are primarily intended for object-oriented languages. Since our input language does not include inheritance, we modeled these through the use of variant records. That is, a supertype was modeled as a variant record with a separate branch for each subtype. All inherited fields are textually included in the subtypes, rather than the supertype. This gives us nearly identical structures to a direct object-oriented implementation, which is essential for our experiment. It is obviously inferior to an object-oriented implementation in many respects that are irrelevant for this experiment.

Gamma et al. [1995] includes six structural design patterns. We did not consider the creational and behavioral patterns because those influence the interface to the types, rather than the structure of the types. Briefly, the Bridge, Composite, and Decorator patterns all involved compound type changes that Tess correctly identified. The Proxy pattern requires user assistance in choosing which rule generated by Tess is correct. The Flyweight and Facade patterns identify a limitation of the current type change model and implementation. The Adapter pattern is not included because there was no sensible example that had the semantics of Adapter without also sharing the structure suggested by the Adapter pattern. The details of the patterns experiment are described below.

11.2.2.1 *The Bridge Pattern*. The purpose of the bridge pattern is to weaken the connection between an abstraction and its implementation. A common mistake made by people new to object-oriented design is to have alternative implementations be subclasses of the abstraction. A difficulty arises when the programmer wants to create multiple related abstractions. Typically, the result is an explosion in the number of subclasses. The Bridge pattern solves this by defining an abstraction whose subclasses are refined abstractions, an implementation whose subclasses are refined implementations, and adding a connection (a bridge) to bind an implementation to an abstraction. Figure 19 shows type definitions before and after the introduction of the Bridge pattern. Figure 20 shows the derivation rule derived for the old *XIconWindow* type. The remaining derivation rules are quite similar. In this case each old object that was an instance of some subclass of Window is broken into two objects, a Window object and a WindowImp object. This is an example of an encapsulation compound change, where part of an old type is encapsulated to form a new type.

11.2.2.2 *The Composite Pattern*. The Composite pattern describes how to organize types representing collections when collections themselves may be considered as parts of a collection. This allows a programmer to develop algorithms that treat collections and components uniformly as much as possible. Figure 21 shows an example before and after the introduction of the Composite pattern. Figure 22 shows the derivation rule for the *Picture* type. This is an example of a merge compound change, because the new

Without the pattern:

```
type Window is
    name: String;
end Window;

type XWindow extends Window is
    gc: GraphicsContext;
end XWindow;

type PMWindow extends Window is
    pm: PMWindowData;
end PMWindow;

type XIconWindow extends XWindow is
    icon: Picture;
end XIconWindow;

type PMIconWindow extends PMWindow is
    icon: Picture;
end PMIconWindow;
```

With the pattern:

```
type Window is
    name: String;
    impl: WindowImp;
end Window;

type IconWindow extends Window is
    icon: Picture;
end IconWindow;

type WindowImp is
    null;
end WindowImp;

type XWindow extends WindowImp is
    gc: GraphicsContext;
end XWindow;

type PMWindow extends WindowImp is
    pm: PMWindowData;
end PMWindow;
```

Fig. 19. Bridge pattern.

XIconWindow ⇒ *IconWindow*: **Compound Change**
new.icon : **derive from** *old.icon*;
new impl = **new** *XWindow*;
new.impl.gc : **derive from** *old.gc*;

Fig. 20. Derivation rule for the bridge pattern.

Without the pattern:

```
type Point is
    x : integer;
    y : integer;
end Point;

type Line is
    point1 : Point;
    point2 : Point;
end Line;

type LineList is sequence of Line;
type PointList is sequence of Point;

type Picture is
    lines : LineList;
    points : PointList;
    pictures : PictureList;
end Picture;

type PictureList is sequence of Picture;
```

With the pattern:

```
type Graphic is
    null;
end;

type Point extends Graphic is
    x : integer;
    y : integer;
end Point;

type Line extends Graphic is
    point1 : Point;
    point2 : Point;
end Line;

type GraphicList is sequence of Graphic

type Picture extends Graphic is
    graphics : GraphicList;
end Picture;
```

Fig. 21. Composite pattern.

```
Picture ⇒ Picture: Compound Change
    new.graphics = new GraphicsList;
    for each line in old.lines
        new_line : derive from line;
        append new_line to new.graphics
    end for;
    for each point in old.points
        new_point : derive from point;
        append new_point to  new.graphics
    end for;
    for each picture in old.pictures
        new_picture : derive from picture;
        append new_picture to new.graphics
    end for;
```

Fig. 22. Derivation rule for the composite pattern.

Without the pattern:

```
type OutputStream is
    fd : FileDescriptor;
    buffer : CharArray (1..100);
    numInBuffer : integer;
    checkSum : integer;
end OutputStream;
```

With the pattern:

```
type OutputStream is
    null;
end OutputStream;

type FileOutputStream extends OutputStream is
    fd: FileDescriptor;
end FileOutputStream;

type Filter extends OutputStream is
    outStream : OutputStream;
end Filter;

type BufferedOutputStream extends Filter is
    buffer : CharArray (1..100);
    numInBuffer : integer;
end BufferedOutputStream;

type CheckedOutputStream extends Filter is
    checkSum : integer;
end CheckedOutputStream;
```

Fig. 23. Decorator pattern.

type, *GraphicsList*, combines the old list types into a single type. The separate list objects are merged into a single list.

11.2.2.3 *The Decorator Pattern*. The Decorator pattern is intended to allow dynamic extension of a class. This is particularly useful in situations where it would be desirable to have multiple extensions simultaneously. Without this pattern, a design either contains a proliferation of subclasses to allow all combinations or a type that contains more information and functionality than may be needed at any time. Figure 23 provides an example without and with the Decorator pattern. Figure 24 shows the mapping from *OutputStream* to *BufferedOutputStream*. An equally valid mapping would go from *OutputStream* to *CheckedOutputStream*. The user needs to decide which mapping is preferred. This example shows several

```
OutputStream ⇒ BufferedOutputStream: Compound Change
    new.buffer : derive from old.buffer;
    new.numInBuffer : derive from old.numInBuffer;
    new.outStream = new CheckedOutputStream;
    new.outStream.checkSum : derive from old.checkSum;
    new.outStream.outStream = new FileOutputStream;
    new.outStream.outStream.fd : derive from old.fd;
```

Fig. 24. Derivation rule for the decorator pattern.

Without the pattern:

```
type Graphic is
    null;
end Graphic;

type Image extends Graphic is
    fileName : String;
    imageImp : Bitmap;
    extent : Rectangle;
end Image;

type GraphicList is sequence of Graphic;

type DocumentEditor is
    images : GraphicList;
end DocumentEditor;
```

With the pattern:

```
type Graphic is
    null;
end Graphic;

type Image extends Graphic is
    fileName : String;
    imageImp : Bitmap;
    extent : Rectangle;
end Image;

type ImageProxy extends Graphic is
    fileName : String;
    extent : Rectangle;
    img : Image;
end ImageProxy;

type GraphicList is sequence of Graphic

type DocumentEditor is
    images : GraphicList;
end DocumentEditor;
```

Fig. 25. Proxy pattern.

```
Image ⇒ ImageProxy : Compound Change
    new.fileName : derive from old.fileName;
    new.extent : derive from old.extent;
    new.img = new Image;
    new.img.imageImp : derive from old.imageImp;
    new.img.fileName : derive from old.fileName;
    new.img.extent : derive from old.extent;
```

Fig. 26. Derivation rule for the proxy pattern.

encapsulation compound changes and the corresponding division of a single object into multiple objects.

11.2.2.4 *The Proxy Pattern.* The Proxy pattern is used to allow a stub object to be a placeholder for a real object until the real object is needed. This can be useful in situations in which the real object is often not needed and is expensive or large to keep in memory. For example, it might require reading a file or communication over a network to get the real value. Figure 25 provides an example without and with the Proxy pattern. Figure 26

Without the pattern:

```
type Glyph is
    null;
end Glyph;

type Row extends Glyph is
    members: sequence of Glyph;
end Row;

type Column extends Glyph is
    members: sequence of Glyph;
end Column;

type Font is
    fontId : integer;
    size : integer;
end Font;

type Char extends Glyph is
    charId : integer;
    charFont : Font;
end Char;
```

With the pattern:

```
type Glyph is
    null;
end Glyph;

type Row extends Glyph is
    members: sequence of Glyph;
end Row;

type Column extends Glyph is
    members: sequence of Glyph;
end Column;

type Font is
    fontId : integer;
    size : integer;
end Font;

type Char extends Glyph is
    charId : integer;
end Char;

type GlyphContext is
    index : integer;
    fonts : BTree;
end GlyphContext;
```

Fig. 27. Flyweight pattern.

shows the mapping from *Image* to *ImageProxy*. This change involves an encapsulation to create the new type and duplication of values between the *Image* type that continues to exist and the new *ImageProxy* type. The derivation rule listed below fits well with the compound type change model described in this paper. The implementation in Tess, however, does not currently support duplication. Therefore, the derivation rule produced by Tess initializes the *fileName* and *extent* fields of the *ImageProxy*, but not the new *Image*. This is an example where user assistance is needed to produce the correct derivation rule.

11.2.2.5 *The Flyweight Pattern*. The Flyweight pattern is an interesting example because it demonstrates a type change that it is difficult to imagine any type evolution system being able to support without significant user involvement. The Flyweight pattern recognizes that it is useful to create new structures to hold small immutable objects and provide a mechanism to look up the correct value rather than hold references to those values in multiple places. Figure 27 shows an example before application of the pattern in which each *Char* object includes which font it uses. After application of the pattern, a *BTree* is created that contains a font object for each font change between consecutive characters. To determine which font to use, it is necessary to know an index for the character in question and then its font can be looked up in the *BTree*. To construct the appropriate

BTree requires significant knowledge about the semantics of the type and the change being performed. We cannot hope to automate this change.

11.3 Value of Exhaustive Search Algorithm

To determine the effectiveness of the various algorithms, we ran Tess with different algorithms disabled. The original results presented in Table II were the result of using the name comparison algorithm, use site algorithm, and exhaustive search algorithm. As the name suggests, the exhaustive search algorithm is expensive computationally. We reran both experiments with the exhaustive search algorithm disabled and found very similar results in the derivation rules produced, but significantly better performance. Recall that with exhaustive search enabled Tess was not able to identify three type deletions in the test suite. With the exhaustive search algorithm disabled, Tess correctly reported that two of the types were deleted. There were no correct derivation rules missed with the exhaustive search algorithm disabled.

11.4 Value of Use Site Algorithm

To determine the effectiveness of the use site algorithm, we ran both experiments again disabling both exhaustive search and the use site algorithms. In this case, all three instances of deleted types in the input were correctly identified as such. In addition, two types were now flagged as deleted when they were transformed to types with different names. This happened in the test case for the Bridge pattern. In this case an old type that was not referenced anywhere was split into two new types, neither of which had the same name as the original type. Since the type's name changed and the old type was not used as a component type anywhere, the name comparison algorithm did not find the appropriate pair of types to compare. The use site algorithm was developed to address this exact situation and thus has demonstrated its usefulness.

11.5 Summary

The type comparison algorithms implemented in Tess performed very well in the experimentation. We found that most type changes are simple ones that are easy to describe with a simple type change model and are accurately recognized by the type comparison algorithms. The algorithms perform extremely well when there are strong naming similarities between the old types and new types and the old field names and new field names. The algorithms also perform well when structural similarities are strong even if the names have changed.

The weaknesses arise when neither naming nor structural similarities are very strong. This typically leads to situations in which it is necessary for the user to disambiguate among a number of equally valid derivation rules. An example of this is when an integer field is replaced with an enumerated type. If symbolic constants were used to represent the values in the integer field and those same names are used as the values in the

enumerated type, then it would be possible to write an algorithm that could produce the correct derivation rule. If integer constants are used, however, a default derivation rule might map the integers beginning with 0 or 1 to the enumerated values in order. It is clear that there are many other derivation rules to choose from and we cannot hope for an algorithm that could find the correct derivation rule without user assistance in general.

We found a few instances of compound type changes in the implementation histories and more instances in the design patterns experiment. As with simple type changes, the more naming similarities that exist with compound type changes the better the type comparison algorithms perform. As the case study in Section 11.1 demonstrates, the compound change algorithms are able to detect nontrivial changes to the types. The opportunity for ambiguity arises again, particularly in situations where a field that holds a single value in the old version holds an array of values in the new version. The ambiguity revolves around where the old value should be placed in the new array. In particular, should it be placed in a single new element? If so, which one? Another possibility is that the old value should serve as the initial value of all the array elements. Obviously, a human user will need to disambiguate this case.

In the course of the experiment, we found one weakness of the type change model. Specifically, it does not allow us to express the need to create a new shared object during transformation. We encountered this limitation when working with the Facade pattern. The Facade pattern introduces an object to centralize communication between two subsystems. This new object must be referenced by all the old objects of one subsystem that used to point into the internals of the second subsystem. While the compound type change model allows new objects to be created during transformation, it does not support the notion of being able to share those new objects. This is clearly a limitation of the model that needs to be addressed in future work.

12. FUTURE WORK

The type comparison algorithms currently operate on a type model that does not have inheritance. The algorithms can still be used with a language that has inheritance by inlining the inherited fields while translating to the internal type model. The advantage of this is that languages with different inheritance semantics can be supported simply by defining the translator that inlines the correct variables according to the source language's inheritance rules. The disadvantage with this solution is that we lose information that would be useful in reducing the number of types that we compare and the amount of work involved in each comparison. By inlining the inherited fields, we duplicate their comparisons in each subtype. Also, we could use the knowledge of supertype-subtype relationships to guide the order in which we do comparisons, much as we currently use the record-component relationships now. This would probably not improve the accuracy of Tess but would almost certainly improve its performance.

The only serious limitation identified with the type change model is the inability to create new shared objects during transformation. Clearly, we must extend the type change model to support this. Also, we would like to evaluate the model and algorithms on the histories of larger and longer-lived systems to further evaluate them.

The type comparison algorithms currently implemented in Tess compare types based upon the structure of those types. Zaremski and Wing have demonstrated the use of type comparison to locate components in a library for reuse [Zaremski and Wing 1995a, 1995b]. The type comparison algorithms that they use rely on type signatures and formal specifications. Since signatures and formal specifications generally change less frequently than representations, incorporating these algorithms into Tess may improve Tess's ability to find matching types in old and new versions of a system. The algorithms to compare types at the representational level are still required to produce the transformers between the types. Using signatures and formal specifications in comparisons may also make it apparent that the database must evolve to respond to changing semantics of the types, even when the representations are unmodified. For example, if a list type is changed from an unsorted list to a sorted list, the representation would not be changed, but the existing values would still not be appropriate to use with the new definition.

Type change is also an issue for dynamic module replacement systems whose goal is to replace program components without stopping execution of a program. In this case there is existing data that may need to be transformed even though it is not necessarily persistent data. Existing systems (such as Fabry [1976], Frieder and Segal [1991]) recognize the need for such transformation functions, but leave the development of those functions to the maintainer. Tess's comparison algorithms could be used to generate these transformation functions.

Another situation in which type comparison may be applicable is schema integration. Here the goal is to develop derivation rules between the types defined in interoperating databases in order that they can share data. In this scenario, the role of the maintainer will become more important as the assumption of naming similarities will most likely be violated. Also, the maintainer would be able to provide valuable guidance in distinguishing between types whose data should be shared and types whose data should remain encapsulated within one database. With the maintainer's guidance, derivation rules could be developed between schemas to allow the necessary data sharing to occur.

Finally, the same issues arise in object serialization supported by Java. Here an object is serialized based upon its type definition in the sender. If the receiver uses a different version of the type, the receiver is unable to unserialize the object, resulting in a runtime exception. This occurs both when reading and writing objects in files and also when sending objects over the network using RMI. The best solution here would be to set up a transformation server so that if a version mismatch is encountered when the object is received, the object and the version number desired by the

receiver could be sent to the transformation server which would transform the object to the desired version. In this case, we would want the ability to transform either to a newer version or an older version. Generating transformers from new versions to old should be no more difficult than from old to new and, perhaps, can be even more fully automated once the derivation rules from old to new have been approved by the user.

13. CONCLUSIONS

During software maintenance, a maintainer is typically expected to increase the functionality of software and improve its performance while maintaining backward compatibility. Backward compatibility is required so that existing users will not need to be retrained to use the new version of the system, and so that existing persistent data can continue to be used. With traditional approaches to managing persistent data, it is typically impractical to make major changes to types for which there is persistent data. This restriction in changing type definitions complicates the design and implementation of the desired functionality and performance modifications.

Our research into persistent type evolution addresses the problem of modifying types for which persistent data exists. Specifically, we have defined a model of type changes that describes the complex type changes we have observed in maintenance histories of real systems. We have developed algorithms to recognize these type changes and to generate derivation rules that can translate data from an old representation to the new representation. By doing so, we offer the maintainer much greater flexibility in the modification of persistent types than traditional database systems do.

ACKNOWLEDGMENTS

Numerous people have been involved in the design and implementation of Tess: Tareef Kawaf, David Maryakhin, Steve Battisti, Jai Shan, Adrian Koren, Heather Conboy, and Yang Wang. I would like to thank Lori Clarke for her support of this work. Additionally, I would like to thank Lori, Peri Tarr, Lee Osterweil, Rick Lerner, and the anonymous referees for their helpful comments on earlier versions of this paper.

REFERENCES

AMADIO, R. M. AND CARDELLI, L. 1993. Subtyping recursive types. *ACM Trans. Program. Lang. Syst. 15*, 4 (Sept.), 575–631.

ARNOLD, K. AND GOSLING, J. 1998. *The Java Programming Language*. 2nd ed. Addison Wesley Java series. ACM Press/Addison-Wesley Publ. Co., New York, NY.

ATKINSON, M., BAILEY, P., CHISHOLM, K., COCKSHOTT, W., AND MORRISON, R. 1983. An approach to persistent programming. *Comput. J. 26*, 4 (Nov. 1983), 141–146.

BANERJEE, J., KIM, W., KIM, H.-J., AND KORTH, H. F. 1987. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM-SIGMOD Conference on Management of Data* (San Francisco, CA, May), ACM, New York, NY, 311–322.

BRATSBERG, S. 1992. Unified class evolution by object-oriented views. In *Proceedings of the 11th International Conference on The Entity-Relationship Approach* (Karlsruhe, Germany, Oct. 7-23), 423–439.

BRECHE, P. 1996. Advanced primitives for changing schemas of object databases. In *Proceedings of the 1996 Conference on CAiSE* (Heraklion, Crete, May).

BRECHE, P., FERRANDINA, F., AND KUKLOK, M. 1995. Simulation of schema change using views. In *Proceedings of the 6th International Conference on Database and Expert Systems Applications* (London, UK, Sept. 1995).

CASAIS, E. 1990. Managing class evolution in object-oriented systems. In *Object Management*, D. Tsichritzis, Ed. University of Geneva, Geneva, Switzerland, 133–195.

CLAMEN, S. M. 1994. Schema evolution and integration. *Distrib. Parallel Databases 2*, 1 (Jan. 1994), 101–126.

DOWLING, G. AND HALL, P. 1980. Approximate string matching. *ACM Comput. Surv. 12*, 4, 381–402.

FABRY, R. 1976. How to design a system in which modules can be changed on the fly. In *Proceedings of the International Conference on Software Engineering* (Los Alamitos, CA), 470–476.

FERRANDINA, F. AND LAUTEMANN, S. -E. 1996. An integrated approach to schema evolution for object databases. In *Proceedings of the Third International Confernce on Object-Oriented Information Systems* (OOIS, London, UK, Dec.), 280–294.

FERRANDINA, F., MEYER, T., AND ZICARI, R. 1994. Implementing lazy database updates for an object database system. In *Proceedings of the 20th International Conference on Very Large Data Bases* (VLDB'94, Santiago, Chile, Sept.), VLDB Endowment, Berkeley, CA, 261–272.

FRIEDER, O. AND SEGAL, M. E. 1991. On dynamically updating a computer program: from concept to prototype. *J. Syst. Softw. 14*, 2 (Feb. 1991), 111–128.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

GARLAN, D., KRUEGER, C. W., AND LERNER, B. S. 1994. TransformGen: automating the maintenance of structure-oriented environments. *ACM Trans. Program. Lang. Syst. 16*, 3 (May 1994), 727–774.

HABERMANN, A N AND NOTKIN, D 1986. Gandalf: software development environments. *IEEE Trans. Softw. Eng. SE-12*, 12 (Dec.1986), 1117–1127.

HABERMANN, N., GARLAN, D., AND NOTKIN, D. 1991. Generation of integrated task-specific software environments. In *CMU Computer Science: A 25th Anniversary Commemorative*, R. F. Rashid, Ed. ACM Press anthology series. ACM Press, New York, NY, 69–97.

JOHNSON, R. E. AND OPDYKE, W. F. 1993. Refactoring and aggregation. In *Proceedings of the International Symposium on Object Technologies for Advanced Software* (ISOTAS '93, Nov.), Springer Lecture Notes in Computer Science Springer-Verlag, New York, NY, 264–278.

KIM, H.-J. AND KORTH, H. F. 1988. Schema versions and DAG rearrangement views in object-oriented databases. Tech. Rep. TR-88-05. University of Texas at Austin, Austin, TX.

KUKICH, K. 1992. Technique for automatically correcting words in text. *ACM Comput. Surv. 24*, 4 (Dec. 1992), 377–439.

LAUTEMANN, S. -E. 1997. A propagation mechanism for populated schema versions. In *Proceedings of the International Conference on Data Engineering* (Birmingham, UK, Apr.), IEEE Computer Society, Washington, DC, 67–78.

LAUTEMANN, S. -E. 1997. Schema versions in object-oriented database systems. In *Proceedings of the 5th International Conference on Database Systems for Advanced Applications* (Melbourne, Australia, Apr.), R. Topor and K. Tanaka, Eds. World Scientific Publishing Co., Inc., River Edge, NJ.

LERNER, B. S. AND HABERMANN, A. N. 1990. Beyond schema evolution to database reorganization. In *Proceedings of the Joint ACM European Conference on Object-Oriented Programming: Systems, Languages, and Applications* (OOPSLA/ECOOP '90, Ottawa, Canada, Oct. 21–25), A. Yonezawa, Ed. ACM Press, New York, NY, 67–76.

LIEBERHERR, K. J., BERGSTEIN, P., AND SILVA-LEPE, I. 1991. Abstraction of object-oriented data models. In *Entity-Relationship Approach: The Core of Conceptual Modelling*, H. Kangassalo, Ed. Elsevier Sci. Pub. B. V., Amsterdam, The Netherlands, 89–102.

MONK, S. AND SOMMERVILLE, I. 1992. A model for versioning classes in object-oriented databases. In *Proceedings of the Tenth British National Conference on Databases* (Aberdeen, Scotland, 1992).

MOORMANN ZAREMSKI, A. AND WING, J. M. 1995. Signature matching: A tool for using software libraries. *ACM Trans. Softw. Eng. Methodol. 4*, 2 (Apr. 1995), 146–170.

MOORMANN ZAREMSKI, A. M. AND WING, J. M. 1995. Specification matching of software components. In *Proceedings of the 17th International Conference on Software Engineering* (ICSE-17, Seattle, WA, Apr. 23–30), D. Perry, Ed. ACM Press, New York, NY.

NAVATHE, S. B. 1980. Schema analysis for database restructuring. *ACM Trans. Database Syst. 5*, 2 (June), 157–184.

ODBERG, E. 1994. MultiPerspectives: The classification dimension of schema modification management for object-oriented databases. In *Proceedings of the 1994 Conference on TOOLS-USA* (TOOLS-USA, Santa Barbara, CA, Aug.),

OPDYKE, W. F. 1991. Refactoring: A program restructuring aid in designing object-oriented application frameworks. Ph.D. Dissertation. University of Illinois at Urbana-Champaign, Champaign, IL.

OPDYKE, W. F. AND JOHNSON, R. E. 1993. Creating abstract superclasses by refactoring. In *Proceedings of the 1993 ACM Conference on Computer Science* (CSC '93, Indianapolis, IN, Feb. 16–18), S. C. Kwasny and J. F. Buck, Eds. ACM Press, New York, NY, 66–73.

PENNEY, D. J. AND STEIN, J. 1987. Class modification in the GemStone object-oriented DBMS. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '87, Orlando, FL, Oct. 4–8), N. Meyrowitz, Ed. ACM Press, New York, NY, 111–117.

PATERSON, J. 1980. Computer programs for detecting and correcting spelling errors. *Commun. ACM 23*, 676–687.

RA, Y. G. AND RUNDENSTEINER, E. A. 1994. A transparent object-oriented schema change approach using view evolution. Tech. Rep.. University of Michigan, Ann Arbor, MI.

RICHARDSON, D. J. 1994. TAOS: Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis* (ISSTA '94, Seattle, WA, Aug. 17–19), T. Ostrand, Ed. ACM Press, New York, NY, 138–153.

SHLAER, S. AND MELLOR, S. J. 1992. *Object Lifecycles:: Modeling the World in States*. Yourdon Press Computing Series. Yourdon Press, Upper Saddle River, NJ.

SHNEIDERMAN, B. AND THOMAS, G. 1982. An architecture for automatic relational database system conversion. *ACM Trans. Database Syst. 7*, 2 (June), 235–257.

SHU, N. C., HOUSEL, B. C., AND LUM, V. Y. 1975. CONVERT: A high level translation definition language for data conversion. *Commun. ACM 18*, 10, 557–567.

SJØBERG, D. I. K. 1993. Thesaurus-based methodologies and tools for maintaining persistent application systems. Ph.D. Dissertation. University of Glasgow, Glasgow, Scotland, UK.

SKARRA, A. H. AND ZDONIK, S. B. 1986. The management of changing types in an object-oriented database. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPLSA '86, Portland, OR, Sept. 29-Oct. 2), N. Meyrowitz, Ed. ACM Press, New York, NY, 483–495.

TARR, P. AND CLARKE, L. A. 1993. Pleiades: an object management system for software engineering environments. *SIGSOFT Softw. Eng. Notes 18*, 5 (Dec. 1993), 56–70.

TRESCH, M. AND SCHOLL, M. H. 1992. Meta object management and its application to database evolution. In *Proceedings of the 11th International Conference on The Entity-Relationship Approach* (Karlsruhe, Germany, Oct. 7-23), 299–321.

WILEDEN, J. C., WOLF, A. L., FISHER, C. D., AND TARR, P. L. 1988. Pgraphite: an experiment in persistent typed object management. *SIGPLAN Not. 24*, 2 (Feb.), 130–142.

ZHU, J. AND MAIER, D. 1989. Computational objects in object-oriented data models. In *Proceedings of the Second International Workshop on Database Programming Languages* (Gleneden Beach, OR, June 4–8), R. Hull, R. Morrison, and D. Stemple, Eds. Morgan Kaufmann Publishers Inc., San Francisco, CA, 139–160.

Received: June 1996; revised: April 1999; accepted: November 1999