Type Evolution Support for Complex Type Changes[*]

Barbara Staudt Lerner
Computer Science Department
University of Massachusetts, Amherst
CMPSCI Technical Report 94–71

October 31, 1994

Abstract

Type evolution is a serious problem for programs that use persistent data. Type changes are required during software maintenance, but can make persistent data inaccessible. Existing systems that support type evolution severely restrict the ways in which a type can be changed, thereby limiting what can be accomplished during maintenance. Tess is a system to automate type evolution for a collection of more complex type changes while providing a more natural editing environment for the programmer. Tess uses a comparative approach to identify type changes rather than a specialized editing process. Experimental results on real data indicate that Tess can accurately identify more complex type changes than those supported by existing systems relying on specialized editing.

1 Motivation

Software maintenance is a serious problem for organizations developing and using software. The maintenance activity is widely recognized as extremely difficult and error-prone, with costs typically exceeding those of initial project development. Software maintenance tools are needed to help manage complexity and to improve reliability of maintained software.

One specific problem encountered during maintenance is unique to programs that manipulate persistent data. The maintenance problem arises when the type definitions corresponding to existing persistent data are modified. The persistent data might be stored in a database, a file system, or managed by a programming language's persistence mechanism directly. In any case, the actual storage representations used by the persistence mechanisms are derived from the structures used in the type definitions. If the structures change, it may no longer be possible to retrieve existing persistent data because it uses an obsolete storage representation.

Because the impact of a change involving a persistent type can be great, designers and programmers may constrain the type changes they will consider to ensure that persistent data remain accessible. If the programmers are provided with tools to help manage the complexities introduced by changing persistent types, they should be less constrained about changing persistent types with the result that they can concentrate on other issues, such as appropriateness of abstractions, modularity, efficiency, etc. While there is no hard data to support this claim, personal experience and anecdotal evidence exist [GKL94, Sjø93b]. Unfortunately, existing tools are extremely limited in their support for type evolution.

[*]This work was supported by the Advanced Research Projects Agency under Grant MDA972-91-J-1009.

While the problem of evolving data representations exists for systems with weak typing mechanisms, such as traditional database systems and information stored in file systems, in such systems the persistent data representations can be localized, and programmers can either resist changing those representations, develop ad hoc mechanisms to update the stored data, or use the mechanisms provided by the database, which typically support a very limited collection of changes. However, with persistent programming languages the problem is more pervasive since one goal of persistent programming languages is to reduce or eliminate the distinction between transient and persistent types. In particular, some persistent programming languages, such as PGraphite, Pleiades, Napier-88, and PS-Algol [WWFT88, TC93, DCBM89, ABC$^+$83], treat persistence orthogonally to type definitions. 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. However, this aggravates the maintenance problem because every type potentially has persistent data associated with it. Modifying any type definition may make some persistent data inaccessible.

Discarding obsolete data is only feasible if the data can be regenerated from some external source. For example, if the data is an abstract syntax tree, it could be regenerated by parsing the source file. However, if the data contains historical information or information provided by an end user, it is unlikely that it could be easily regenerated. Even when regeneration is possible, it could be quite costly. For example, if the persistent data represents the parse trees for a million line program, regeneration will be quite time-consuming. It is therefore imperative that the existing data remain accessible.

Requiring that type definitions remain constant over time is also impractical in the general case. Maintenance involves adding functionality, improving performance, etc., which generally requires not just adding new types, but also modifying existing types. It is therefore imperative that programmers be able to modify types.

There are two approaches to keeping obsolete data accessible after types change. Either the data can be changed to conform to the type's new representation or the procedures responsible for reading and writing persistent data can be made flexible enough to handle multiple representations. In either case, a primary goal is to isolate the application code from concerns about encountering data that uses an obsolete representation. We call the code that manages a type's changing representations a transformer, whether the transformation updates the persistent memory or maintains multiple representations.

Developing a transformer by hand is cumbersome and error prone. It requires the ability to refer to multiple versions of a type within the same program. Since types are frequently modified without changing their names, straightforward implementation of a transformer would result in multiple definitions for the same type name. This therefore necessitates extending the typing system to incorporate versioning information. If a programmer is responsible for creating and maintaining the transformer, this extension to the typing system must be made available to the programmer, significantly complicating the typing model. It would be better to not only isolate the application code, but also the application developer, from the implementation details of transformers.

Putting these requirements together leads to the conclusion that programmers must be able to modify types, existing data must remain accessible, and automated assistance in the development of a transformer is both appropriate and desirable. Indeed, some database systems offer support for type or schema evolution. However, they typically support only simple type changes isolated to individual types, such as adding records to fields. We extend support to more complex type changes, in particular those that cannot be described accurately using only changes isolated to individual types. These include such changes as moving a field from one record to another, dividing a type into multiple types, or merging multiple types into a single type.

We have built a system called Tess (Type Evolution Software System) that compares sets of type definitions to produce a transformer. A naive implementation of such a system would require an exponential number of type comparisons to occur, as we would need to compare each combination of old types with each combination of new types. In this paper, we describe the algorithms that we use to reduce the number of comparisons performed in practice to a feasible level. We also describe the basic nature of the algorithms that perform the type comparison and demonstrate their behavior on an interesting example of type evolution occurring in the course of maintaining a real project.

2 Related Work

The problem of persistent type evolution was first addressed with respect to traditional database systems, where it is known as schema evolution. While many database systems support a few simple changes automatically, such as adding or deleting record fields, only a few systems [SHL75, Nav80, ST82] support more general transformations. In these cases, the programmer 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, but there is little tool support to assist in this process.

Several researchers have investigated type evolution in object-oriented databases. Orion [BKKK87, KK88], GemStone [PS87], and O_2 [Bar91] are object-oriented database systems that provide some evolution support. In these systems, evolution is defined in terms of primitive structural changes isolated to individual type definitions, such as adding instance variables to a class, removing instance variables from a class, and renaming instance variables. More complex type changes, such as combining two records, are not supported directly. Instead this change is accomplished as several independent primitive changes involving the addition of new instance variables to one of the types and the deletion of the other type. The result is that the data associated with the deleted type is lost and the instance variables added to the remaining type are uninitialized. To preserve the data, the programmer must develop code to move the data explicitly. Some simple changes are completely automated, but at the expense of further limiting the ways in which a programmer 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; it cannot be replaced by a subtype, nor can it be replaced by an arbitrary type in the type hierarchy.

Another approach to evolution of object-oriented databases relies on the simultaneous maintenance of multiple versions of classes and objects. This approach has been used by Skarra and Zdonik [SZ86], Clamen [Cla93], and Bratsberg [Bra92]. With these approaches, multiple versions of the same class exist within a single database. The advantage is that existing and new code can operate on existing and new objects without requiring either to be changed. The disadvantage is that the programmer must provide routines to make objects appear to be of the version of a class that the code is expecting. This approach admits more general type changes, but it still limits type changes to be isolated to individual types. It also results in significant overhead (in both space and time) for maintaining and accessing multiple class and object versions. Odberg [Odb94] extends the versioning approach to the entire class hierarchy, which is versioned when a type is modified. This allows the description of type changes that simultaneously affect multiple types, but results in a complicated type system that would be difficult to maintain as the number of versions increases.

TransformGen [GKL94] is a system to support evolution of abstract syntax grammars used by Gandalf programming environments [HN86, HGN91]. 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. However, TransformGen goes beyond these two systems by allowing the programmer to modify the generated transformation functions.

The significance of this extensibility is that the resulting transformers can handle arbitrary type changes, including those involving multiple types. However, while the programmer can extend the transformer, there is little guidance in identifying the limits of the generation process and the situations that require extension. OTGen [LH90] 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 of TransformGen, but is aimed at a more general type system.

3 Type Comparison

The fact that persistent data exist should not constrain the type changes that a programmer makes. Neither what changes are made nor the editing process with which those changes are made should be constrained. Existing systems with automated evolution support constrain the editing process. They require the programmer to specify the type changes explicitly. For example, in Orion rather than using a text editor to modify type definitions, the programmer uses a special set of commands that explicitly identify the type change, such as adding an instance variable. In TransformGen, the same (structure) editor is used to modify grammars as to create them. However, during modification, the editing commands have additional evolution semantics associated with them. When a grammar production is modified, the editing command used to modify the production determines the type of transformation that is required to update the persistent data created with that production. In this case, the system behaves as if it is explicitly told what the change was, although from the programmer's viewpoint the system appears to infer the evolution effect.

There are several problems with the explicit editing approach characterized by Orion and TransformGen. First, the programmer must either learn and use a different set of commands when modifying type definitions than when creating them or understand the additional evolution semantics associated with the editing commands.

Second, the programmer must understand that the editing *process* defines the evolution semantics, not the editing *result*. For example, in TransformGen a term can be added to a production[1] by duplicating an existing term using cut-and-paste to cut the term and paste it twice or copy-and-paste to copy the term and paste it once. Both editing processes have the same editing result. However, they have different evolution semantics. With cut-and-paste, the original cut has the undesired effect of deleting the term *and* its associated values. Copy-and-paste leaves the initial term and its values intact. Thus a subtle difference in editing actions can have a significant effect on evolution.

A third related problem is that since restoring syntax involves using editing operations, reversing the syntactic effects of a change by applying editing operations does not necessarily reverse the evolution semantic effects. Cut-and-paste, as described above, is one example of this. A cut immediately followed by a paste in the same location results in no syntactic change, but has significant (and presumably undesired) evolution semantics. As a result, systems based on explicit editing require sophisticated mechanisms to undo both the syntactic effects and evolution semantics associated with editing commands. One approach is to provide undo commands knowledgeable of the syntactic effects and evolution semantics of the editing commands so that they can both be undone. Another approach manifested in TransformGen is to provide a command to revert a production to its previous definition, effectively undoing all changes made to that production. Orion takes a different approach and requires that programmers modify type definitions within transactions. If the programmer wishes to undo some type changes, the entire transaction must be aborted, potentially including changes the programmer doesn't wish to undo. It also limits the scope within which

[1] This is analogous to adding a field to a record or an instance variable to a class.

changes can be undone. With either approach, the programmers must know the importance of using undo or aborting a transaction rather than restoring syntax via editing commands.

Our research has focused on the question of identifying type changes by examining the editing result rather than the editing process. We have developed Tess to demonstrate the feasibility of this approach. Given two sets of type definitions, one representing a newer version of the other, Tess compares the type definitions to identify corresponding types and to define mappings that transform instances of those types from the old representation to the new. Tess's comparison algorithms identify simple structural changes, including adding, deleting, and renaming record fields, changing array bounds, changing record field and array element types, and renaming types. Preliminary experimentation with the structural comparison approach has indicated that it is effective at supporting changes similar to those used in systems based on an explicit editing model. Furthermore, we are able to recognize type changes that simultaneously involve multiple types, such as moving parts of a type definition from one place to another and encapsulating parts of a type definition to create a new type. By allowing programmers to continue to use their normal editors and without requiring deeper understanding of the editing commands, programmers can focus their energies on obtaining the desired editing results, and not on the editing process used to obtain those results, while still getting the benefits of automatic transformer generation.

We make some simplifying assumptions. We assume that code always expects the newest version of a type. By making this assumption we do not need to be concerned with data whose version is newer than what code is expecting. This might not be a reasonable assumption in a large software system in which persistent data is shared by several programs. We expect it is feasible to extend Tess to handle this situation, by creating transformation functions to do the reverse mapping, but we do not currently do so.

Second, 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 programs tend to experience evolutionary change, rather than revolutionary change, we do not expect this to be a significant problem for most situations. Furthermore, we are working on an interactive version of the system in which the programmer can offer guidance to Tess. This guidance is expected to be particularly useful in those situations in which this assumption is violated.

3.1 The Type Comparison Control Algorithm

The input to the type comparison algorithm is the abstract syntax trees representing the type definitions of consecutive versions of a program or module. The type comparison algorithms selectively compare the types to identify how the types have changed and output a description of how to transform instances of the old version into instances of the new version.[2] The 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. Because Tess compares abstract syntax tree versions of the type definitions, it is not affected by changes to white space and comments, or the order in which the type definitions appear in the file.

The type comparison control algorithm proceeds through four stages. First, in the *name comparison* phase, old and new types that have the same names in both versions are compared. For structured types, such as records and arrays, this may result in further type comparisons. For example, in order to understand how to transform an instance of an old array type to an instance of a new array type, we need to know how to transform the elements of the old array as well. Comparing these element types is the second

[2]Ultimately, we will output code to perform the transformations, but that is not implemented yet.

```
type old_record is record                    type new_record is record
    field1: old_field1_type;                     field1: new_field1_type;
    field2: field2_type;                         field2: field2_type;
end record;                                  end record;
```

Figure 1: Component and Use Site Comparisons

phase of the algorithm, called *component comparison*. In the third phase, called *use site comparison* we consider the transformations that we have already identified and analyze types that use types involved in those transformations. In the final phase, called *exhaustive comparison*, we look at each old type. For any old type that does not already have a good transformation to some new type, we exhaustively compare it to each new type, first considering only those new types that use the same type constructor and if that fails, we consider all remaining new types. The exhaustive comparison algorithm contains within it an iteration of the second and third phases. Thus if a good transformation is found by exhaustive comparison, we immediately consider pairs of types used by the matched type pair as well as pairs of types using the matched type pair. This further reduces our search for matching types.

To better understand the relationship of the algorithms, consider the type definitions in Figure 1. *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. However, the two versions of *field2_type* (not shown) are compared in this phase. Assuming Tess finds a mapping between these types, the use site comparison phase 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*, Tess compares *old_field1_type* to *new_field1_type* as a component comparison.

It should be evident that the less the type definitions change the fewer comparisons Tess makes. If Tess compares two identical files, no type names have been changed, and it will find the identical types by comparing exactly one old type with exactly one new type. In the pathological case in which all type names change, Tess does not do any comparisons until the exhaustive comparison phase. At that point, the number of comparisons that it does to identify the matching types depends on how interrelated the types are, whether type constructors have changed, whether record field names have changed, and also how lucky Tess is in the order in which the type comparison proceeds. With Tess we can selectively turn off various phases of the comparison algorithm and observe the effects. We comment on experimentation with this capability in Section 4.2.

3.2 Recognizing Simple Type Changes: A Sample Algorithm

Simple type changes are those changes that affect a single old type and a single new type. When looking for simple changes between two types, the algorithm that Tess uses depends upon the type constructors that the types use. For example, Tess uses a different algorithm to compare two enumerated types than to compare two record types. Tess also has algorithms to compare two types that use different type constructors. For instance, to compare a record to an array, Tess compares the record type to the array element type to determine if a single value should be replaced with a sequence of values. Tess also compares each record field type with the array type to determine if the type change involves saving just one field of the record and discarding the rest.

Here we describe 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 mapping between those records, such that each record field of the old type is either mapped to a record field of the new type or is identified as being deleted, and each record field of the new type is either set to the value of a record field from the old type, initialized to the value in its declaration, or explicitly identified as uninitialized.

The record comparison algorithm is quite similar to the algorithm used to compare the sets of type definitions. First, Tess compares record fields with the same name. Next, it compares old unmatched fields with new unmatched fields with the same type name. Finally, it compares each old unmatched field to each new unmatched field. When it compares fields, it compares the field names and recursively compares the field types. Tess cache the results of type comparisons in a matrix so that it can look up the results of previous comparisons instead of repeating them.

Tess associates a similarity measure with each mapping that it creates. The similarity measure is used to indicate the general nature of the change described in that mapping so that Tess can easily compare mappings to determine which is least severe, both in terms of its impact on persistent data and on the code that manipulates the type. Mappings that represent less severe changes are considered preferable to those that have a greater impact. The measures that a record-to-record mapping can have are the following, listed in order of least severe to most severe:

Identical All the fields of the old record and of the new record have the same field names, the same field type names, and appear in the same order.

Ordering Change All the fields of the old record and of the new record are paired up by matching both the field names and the field type names, but the fields do not appear in the same order in the two records.

Field Type Name Change All the fields of the old record and of the new record are paired up by matching the field names, but one or more of the field's types have changed names.

Field Name Change All the fields of the old record and of the new record are paired up, but the names of one or more fields have changed.

Uninitialized New Field All the fields of the old record match a field of the new record, but some fields in the new record have no match from the old record.

Deleted Old Field Some, but not all, fields of the old record match a field of the new record.

Incompatible No fields of the old record could be mapped satisfactorily to any fields in the new record.

As the above list indicates, the least severe impact is associated with those type changes that neither create nor delete data, although they may result in changes to the storage format of the data or to code manipulating the type. The most severe impact is associated with type changes that would result in the deletion of data. This is most severe since the data may be irretrievably lost. We also consider it unlikely to be the programmer's intent since software maintenance rarely results in decreasing the functionality, and hence the data requirements, of a system.

Using algorithms such as the one described here Tess can recognize changes equivalent to those supported by object bases that provide automatic support for type evolution using editing commands, including Orion, GemStone, and TransformGen.

```
procedure CheckForNesting (RecordMap: in out Mappings.Mapping) is
begin
    FieldsToNest := the unmapped fields of the old type in RecordMap
    for each field in the new type in RecordMap loop
        if the existing mapping to that field has the similarity Uninitialized New Field then
            compare FieldsToNest to the unmapped fields of the new field recursively
        end if;
    end loop;
end CheckForNesting;
```

Figure 2: Algorithm to Recognize the Nesting Compound Change

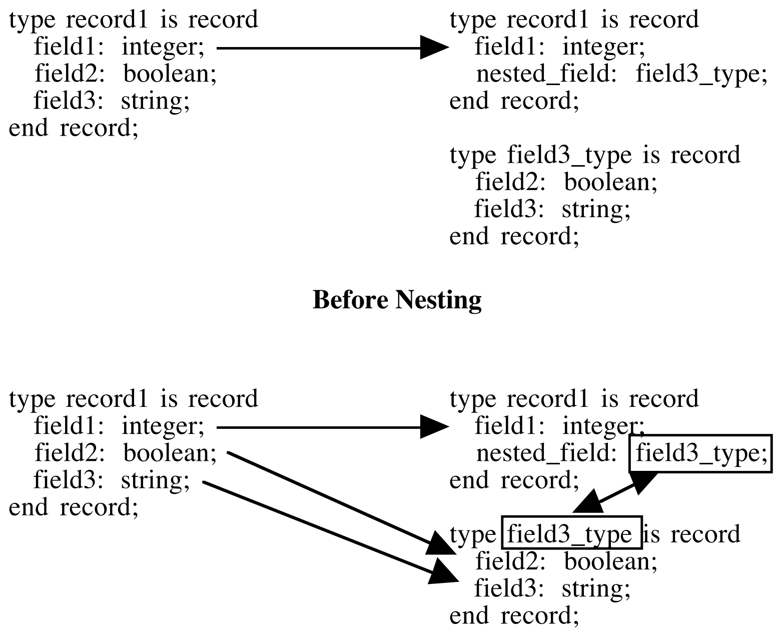
3.3 Recognizing Compound Type Changes: A Sample Algorithm

The real strength of Tess lies in its ability to recognize compound changes, changes that simultaneously affect multiple types. Compound changes are difficult to manage with editing commands because of their variety and complexity. In particular, consider a change in which a field is moved from one record type to another. It would be relatively easy to provide an editing command to modify the types appropriately. The difficult part is understanding how that change should affect the persistent data. Should we create a new instance of the record that the field is moved to? Or should we find an existing instance of the record and add that field to it? If there is more than one instance of the destination record type, how do we determine which value to move to which instance?

Rather than create a collection of subtly different editing commands to move fields, we considered the situations in which this type of change might occur to see if there was a way of recognizing the change by comparing the types. One way in which the change might occur is to move a field from a sub-record to an enclosing record. We call this *unnesting*. Moving from an enclosing record to a sub-record, called *nesting*, is another possibility. Combinations of these are also possible, effectively allowing a field to move from one sub-record to a sibling sub-record.

The algorithms to detect nesting and unnesting are similar. They keep track of fields from old and new record types that are not mapped. Unmapped fields from old types correspond to those from which data can move, while unmapped fields from new types represent those to which data can move. Any field movement recognized as a nesting or unnesting naturally indicates how the data should move between the record instances.

Figure 2 shows the algorithm used to recognize nesting. It is passed a record mapping that describes the transformation between an old and new record type. Its goal is to improve the mapping by nesting unmapped fields of the old type to fields of unmapped fields of the new type. It begins by identifying the fields that are unmapped. It then considers each unmapped new field in turn and attempts to map the old unmapped fields to the fields of the unmapped new field to accomplish the nesting. If necessary, this comparison recursively attempts to nest at the next deeper level. Figure 3 shows the mappings that the nesting algorithm finds for a particular set of type definitions.



```
type record1 is record              type record1 is record
   field1:  integer;                   field1:  integer;
   field2:  boolean;                   nested_field:  field3_type;
   field3:  string;                 end record;
end record;
                                    type field3_type is record
                                       field2:  boolean;
                                       field3:  string;
                                    end record;
```

Before Nesting

```
type record1 is record              type record1 is record
   field1:  integer;                   field1:  integer;
   field2:  boolean;                   nested_field:  field3_type;
   field3:  string;                 end record;
end record;
                                    type field3_type is record
                                       field2:  boolean;
                                       field3:  string;
                                    end record;
```

After Nesting

Figure 3: Nesting Example

4 A Case Study in the Application of Tess

We are experimenting with Tess by looking at examples taken from papers on type evolution as well as examples taken from the maintenance histories of real systems. In this section we present an example taken from a real system involving a complex type change that requires data to move from one record field to another.

4.1 A Compound Type Change Example

There is little published data on how types change through a system's lifetime. It is therefore difficult to know with certainty what evolution support is most beneficial. Sjøberg has published the change history of a relational database management system that includes interesting compound changes like new relations created by joining or partitioning existing relations [Sjø93a]. Additionally, we have access to the maintenance histories of several research systems, which we are using to understand the general nature of evolution, to experiment with Tess, and to guide future development.

To demonstrate the complexity of compound changes handled by Tess, Figures 4 and 5 show consecutive versions of a collection of interrelated types extracted from TAOS, a software testing tool developed at the University of California at Irvine by Debra Richardson and her students[Ric93]. (TAOS is written in Pleiades, an extension of Ada that supports persistence. To understand this example, it is sufficient to consider relationship types to be record types and group types to be union types.) In this example, we see three modified types and four new types. The reader is encouraged to take a few minutes to analyze the example to see that the type changes are non-trivial.

The type changes that occur involve moving fields from the *RandomTestInfo* relationship into the *TestCasesInfo* relationship. Furthermore we see that the new *TestCasesInfo* relationship contains only two fields, both arrays. The change requires unnesting from the *ExtraInfo* field of the old version of the *TestClass* relationship and nesting into the *TestSetInfo* field of the new version of the *TestClass* relationship. To further complicate the example, the type of the *ExtraInfo* field is a union type. Of the three union members, only *RandomTestInfo* contains information that can be nested into the *TestSetInfo* field. If an old instance of the *TestClass* relationship contains either *ManualTestInfo* or *StructuralTestInfo*, we must initialize the *TestSetInfo* field to its declared initial value. This requires us to develop a transformation that includes a conditional to determine how to set the *TestSetInfo* field.

Figure 6 shows the mappings that Tess generates for the *SaveTestCases* and *TestClass* types. For the *TestClass* relationship, we see that Tess correctly identifies that the *Persistence*, *NumberNonPersistentFailed*, and *NumberNonPersistentPassed* fields should be unnested from the *ExtraInfo* field and nested into the *TestSetInfo* field. Tess correctly determines that the mapping from the old *SaveTestCases* enumerated type to an array involves nesting into the array. It also correctly determines that the old *NumberNonPersistentPassed* and *NumberNonPersistentFailed* fields of *RandomTestInfo* should be nested into the new *NumTestCases* field of *TestCasesInfo*, which is also an array. Furthermore, Tess correctly identifies that this mapping is only applicable when the *ExtraInfo* field is actually a *RandomTestInfo* instance, and in other cases the declared initial value should be used.

While Tess does quite well with this example, it does not handle it completely correctly. When nesting values into arrays, Tess does not know where to put the values in the array. This limitation results in two inaccuracies in the TAOS mapping. First, *SaveTestCases* is modified from an enumerated type of two values into an array of booleans. Mapping the enumerated values onto the boolean values is done correctly. The failure is that the old value should be duplicated in each element of the new array rather than just placed

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

type RandomTestInfo is relationship
    Creator : UserSuppliedName := DynamicStrings.Create("Random Test Generator");
    ContextFreeGrammar : FileName;
    MinLength: natural := 0;
    MaxLength: natural := 0;
    NumberRequired: positive := 1;
    Persistence: SaveTestCases := todo;
    NumberNonPersistentFailed: natural := 0;
    NumberNonPersistentPassed: natural := 0;
end relationship;

type ExtraTestInfo is group of (ManualTestInfo, RandomTestInfo, StructuralTestInfo);

type TestClass is relationship
    ClassName : UserSuppliedName;
    ClassID : natural := Get_Unique_Id;
    Project : UserSuppliedName;
    Component : UserSuppliedName;
    Rationale : DynamicStrings.DynamicString;
    Executable : FileName := NullFileName;
    ClassOracle : Oracle;
    Measure : Measurements := ( others => false );
    OutofDate : Boolean := false;
    Generated : Calendar.Time := Calendar.Clock;
    Updated : Calendar.Time := Calendar.Clock;
    TestSet : TestCaseSet := Create;
    ExtraInfo: ExtraTestInfo;
    ClassKind : derived TestClassKind := Get_TestClassKind (ExtraInfo);
end relationship;
```

Figure 4: Old Version of Types Extracted from TAOS

```
type RandomTestInfo is relationship
    Creator : UserSuppliedName := DynamicStrings.Create("Random Test Generator");
    ContextFreeGrammar : FileName;
    MinLength: natural := 0;
    MaxLength: natural := 0;
    NumberRequired: positive := 1;
end relationship;

type TestCaseState is (Pass, Fail, Untested);

type ExtraTestInfo is group of (ManualTestInfo, RandomTestInfo, StructuralTestInfo);

type Saved is ( persistent, nonpersistent );

type SaveTestCases is array ( TestCaseState ) of boolean;

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

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

type TestClass is relationship
    ClassName : UserSuppliedName;
    ClassID : natural := Get_Unique_Id;
    Project : UserSuppliedName;
    Component : UserSuppliedName;
    Rationale : DynamicStrings.DynamicString;
    Executable : FileName := NullFileName;
    ClassOracle : Oracle := NullOracle;
    Measure : Measurements := ( others => false );
    OutofDate : Boolean := false;
    Generated : Calendar.Time := Calendar.Clock;
    Updated : Calendar.Time := Calendar.Clock;
    TestSet : TestCaseSet := Create;
    TestSetInfo : TestCasesInfo := Create;
    ExtraInfo: ExtraTestInfo;
    ClassKind : derived TestClassKind := Get_TestClassKind (ExtraInfo);
end relationship;
```

Figure 5: New Version of Types Extracted from TAOS

```
SaveTestCases => SaveTestCases: Requires New Element
    Element indexed by Pass: See the mapping from SaveTestCases to boolean
    Elements indexed by Fail and Untested will be uninitialized.

SaveTestCases => boolean: Value Change
    Handle each value as follows:
        nada => false
        todo => true

TestClass => TestClass: Requires New Field
    Handle each field as follows:
        ClassName => ClassName
        ClassID => ClassID
        Project => Project
        Component => Component
        Rationale => Rationale
        Executable => Executable
        ClassOracle => ClassOracle
        Measure => Measure
        OutOfDate => OutOfDate
        Generated => Generated
        Updated => Updated
        TestSet => TestSet
        ExtraInfo => ExtraInfo
        ClassKind => ClassKind
        if type (ExtraInfo) = RandomTestInfo then
            ExtraInfo.Persistence => TestSetInfo.PersistencePreferences
            ExtraInfo.NumberNonPersistentFailed => TestSetInfo.NumTestCases(persistent, Pass)
            ExtraInfo.NumberNonPersistentPassed => TestSetInfo.NumTestCases(persistent, Fail)
            The following new fields are initialized to the listed default values:
                TestSetInfo.NumTestCases(persistent, Untested) := Default_Counts (persistent, Untested)
                TestSetInfo.NumTestCases(nonpersistent, Pass) := Default_Counts (nonpersistent, Pass)
                TestSetInfo.NumTestCases(nonpersistent, Fail) := Default_Counts (nonpersistent, Fail)
                TestSetInfo.NumTestCases(nonpersistent, Untested) := Default_Counts (nonpersistent, Untested)
        else
            TestSetInfo := Create
```

Figure 6: The Mapping Generated by Tess for the TAOS Example

in the first element. The second failure deals with the initialization of the *NumTestCases* field of the new *TestCasesInfo* relationship. In the old version of the types we have two fields of type natural. These fields are replaced with an array containing six naturals. As in the previous situation, Tess correctly determines that the old fields should be placed in the array. However, it again has difficulty determining where they should go in the array. Tess uses the first two elements of the array, and initializes the remaining to the declared initial value. However, from reading the names of the old fields and the enumerated values that serve as indices into the array, it is clear that the array is not correctly initialized. Note that in this case duplicating the old information is not appropriate. Instead selecting the appropriate array elements for initialization is the key.

This example demonstrates Tess's flexibility in recognizing simple and compound changes. It also demonstrates a limitation of Tess, its willingness to guess when there is not enough information to make an accurate decision. One direction that we are currently pursuing to improve this situation is to develop an interactive user interface for Tess. With this interface, Tess can create the obvious mappings automatically, while bringing unclear situations to the programmer's attention for disambiguation.

4.2 Performance Results

Tess has been instrumented so that we can evaluate the effectiveness of the various phases of the control algorithm. It can be run in a variety of configurations to see how effective the different algorithms are at pruning the search and how much quality is sacrificed in the mappings produced. The complete TAOS example has 36 types in its old version and 39 types in its new version. Four of the old types are modified, while three new types are added. If no pruning were done at all, Tess would produce $36 \times 39 \times 3 = 4212$ mappings.

Figure 7 shows the results of running Tess on the TAOS example in varying configurations.[3] Using all phases of the algorithm and pruning its search, Tess finds 1443 mappings and reports 37. The unreported mappings have a similarity measure that indicate they are inferior to the reported mappings and are therefore hidden from the programmer. The only inaccuracies in the reported mappings are those outlined in Section 4.1.

Now let's consider some other configurations of the system. If we disable use site comparison, the same mappings are found and reported. This isn't too surprising for this example. None of the type changes involve changes to the type names and all the new types are subordinate types, not users of the old types.

If we disable name comparison, 1521 mappings are found and the same 37 are reported. If we enable only exhaustive comparison, 1638 mappings are found with 165 being reported. Furthermore, not all of the correct mappings are reported. Using only exhaustive comparison results in pruning based only on similarity measures. Pruning done in this way appears to be inferior to that done in conjunction with name matching and use site matching, but Tess does not really benefit from having both name matching and use site matching for this example.

If we turn off exhaustive comparison, Tess identifies 422 mappings, reporting the same 37. The pruning is even better if we inhibit both use site comparison and exhaustive comparison. In this case a mere 40 mappings are identified, yet the same 37 are reported. These results are certainly due to the constancy of the type names in the two versions of the system. We expect that different examples would result in different performance from the various phases and we are continuing experimentation to observe its behavior with other examples.

[3] Note that we never disable the component comparison since it is required to complete mappings, not to initiate new mappings.

Name Comparison	Use Site Comparison	Exhaustive Comparison	Mappings Found	Mappings Reported	Mappings Correct
▨	▨	▨	1443	37	35
▨	▨		422	37	35
▨		▨	1443	37	35
	▨	▨	1521	37	35
▨			40	37	35
	▨		0	0	0
		▨	1638	165	30

▨ Enabled

Figure 7: Experimental Results with Varying Configurations

5 Future Work

We recognize that no matter how sophisticated the type comparison algorithms become, there will always be changes whose correct interpretation defies automation. The ultimate goal of Tess, therefore, is to provide semi-automated assistance in developing sophisticated transformers for persistent data. Automation should generate transformation functions when possible, but allow easy review, modification, and extension by the programmer. In addition, it should support the programmer by focusing his/her energies on the difficult transformations and by type checking the programmer-enhanced transformers functions to reduce programmer errors. Experimentation thus far has demonstrated that the use of similarity measures to distinguish changes that can be automated from those that require the programmer's attention works well.

One problem with developing transformers by hand or extending transformers is that the process is error prone. Unless the transformer is analyzed to assure correctness, errors in the transformer will not be detected until the transformer is applied to a persistent object base. If there is exactly one object base for each collection of type definitions, this may be satisfactory, because we may require the person developing the transformer to also run the transformer and fix any problems encountered. However, if many object bases are created using the same type definitions, or the type definitions are reused in many programs, it is more likely that the data will be transformed in a lazy fashion. When it is accessed and determined to be out-of-date, the transformer will be invoked to update it. Any errors encountered during transformation will be reported to the end-user who should be unaware of the transformation process. To avoid reporting errors to the user, it is essential that the transformer be complete and correct.

There are several aspects to completeness. First, the transformer must be structurally complete. There must be some way of transforming an instance of each old type to some new type(s). If an old type is deleted, there should be some mapping of its data to new types or an explicit statement that the data should be deleted.

Second, each mapping must be complete. Mappings are typically defined in terms of other mappings. For instance, the mapping from one record type definition to another might state that the value from an old record field should be assigned to a new record field. If those record fields use the same unmodified type, this is a simple assignment. However, if different types are used or the type has changed, the record mapping requires a specific mapping from the old field type to the new field type. This is a stronger requirement than the one mentioned previously, because it requires a specific new type to be the destination of the mapping. A tool can keep track of these requirements and ensure that they are satisfied before considering a transformer to be complete. As part of the guidance that Tess can offer in an interactive interface, it will inform the programmer of any incompleteness in the transformer, so that he/she can provide the missing transformation functions.

Another interesting problem of persistent type evolution is evident when considering the richer type models supported by persistent programming languages. In particular, types are typically defined as abstract data types. This emphasizes the semantic properties of the type and de-emphasizes the structural properties of the type. Changing the semantic properties of a type can affect the semantic correctness of existing persistent data, just as changing the type's structure can affect its syntactic correctness. For example, a collection of items might be stored in an array structure. An evolution of this type from an unsorted array to a sorted array is not captured by a change to the structure of the type, but rather by a change to the semantics of the type. It is equally important to ensure that data correspond to new semantic constraints of types as to their syntactic constraints. While capturing the concept of semantic change is not something we expect to be able to automate, we believe that by providing support for transformer extensibility, we can provide the core of the system to update the persistent data, allowing the programmer to focus on implementation of the specific semantic change. Currently O_2 is the only system that considers the semantic effects of evolution. Their emphasis is on how changes to the structure of a class affect the methods in that class, and also how changes to methods affect other methods. They do not consider how changes to a type's semantics affect persistent data.

Another direction we are interested in pursuing is applying type comparison to software re-engineering. In particular, by comparing the types within a single version of a software system, we should be able to identify similarities among the types that could be used to improve their organization, either by creating new types to maintain common information, or by creating an object-oriented type hierarchy.

6 Conclusions

Software maintenance is a difficult software engineering activity. It is important to provide software engineers with tools to assist in maintenance to encourage that changes be done in the most appropriate way, not necessarily the most convenient way. One goal of Tess is to encourage appropriate maintenance by offering the software engineer great flexibility in modifying types, automating type evolution for situations that can be clearly inferred from the type changes, and allowing extensibility of the generated transformer to handle unclear situations. In Tess, we have demonstrated that the comparative approach to recognizing type changes is as effective as an explicit editing approach at interpreting simple changes, and can, in fact, recognize more complex type changes than existing systems. Furthermore, it is not necessary for the programmer to use a special tool to perform the simple changes handled by existing systems. Instead, he/she is only burdened with more effort when complex type changes are made.

References

[ABC$^+$83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4), 1983. Also published in *Readings in Object-Oriented Database Systems*, Stanley B. Zdonik and David Maier, eds., Morgan Kaufman, San Mateo, California, 1990.

[Bar91] G. Barbedette. Schema modifications in the LISPO$_2$ persistent object-oriented language. In P. America, editor, *Proceedings of ECOOP '91, the Fifth European Conference on Object-Oriented Programming*, number 512 in Lecture Notes in Computer Science, Geneva, Switzerland, July 1991. Springer-Verlag.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD 1987 Annual Conference*, pages 311–322, San Francisco, May 1987.

[Bra92] S.E. Bratsberg. Unified class evolution by object-oriented views. In *Proceedings of the 11th International Conference on the Entity-Relationship Approach*, pages 423–439, Karlsruhe, Germany, October 1992. Published as Lecture Notes in Computer Science 645, Springer-Verlag.

[Cla93] Stewart M. Clamen. Schema evolution and integration. *Distributed and Parallel Databases: An International Journal*, 1(4), Dec 1993.

[DCBM89] Alan Dearle, Richard Connor, Fred Brown, and Ron Morrison. Napier88—a database programming language? In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 179–195. Morgan Kaufmann, 1989.

[GKL94] David Garlan, Charles W. Krueger, and Barbara Staudt Lerner. TransformGen: Automating the maintenance of structure-oriented environments. *ACM Transactions on Programming Languages and Systems*, 16(3):727–774, May 1994.

[HGN91] Nico Habermann, David Garlan, and David Notkin. Generation of integrated task-specific software environments. In Richard F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, Anthology Series, chapter 4, pages 69–97. ACM Press, Reading, Massachusetts, 1991.

[HN86] A. Nico Habermann and David Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[KK88] Hyoung-Joo Kim and Henry F. Korth. Schema versions and DAG rearrangement views in object-oriented databases. Technical Report TR-88-05, University of Texas at Austin, February 1988.

[LH90] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In *Proceedings of the Joint ACM OOPSLA/ECOOP '90 Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 67–76, Ottawa, Canada, October 1990.

[Nav80] Shamkant B. Navathe. Schema analysis for database restructuring. *ACM Transactions on Database Systems*, 5(2):157–184, June 1980.

[Odb94] Erik Odberg. MultiPerspectives: The classification dimension of schema modification management for object-oriented databases. In *Proceedings of TOOLS-USA '94*, Santa Barbara, California, August 1994.

[PS87] D. Jason Penney and Jacob Stein. Class modification in the GemStone object-oriented DBMS. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 111–117, Orlando, Florida, October 1987.

[Ric93] Debra J. Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA) '94*. ACM Press, June 28-30 1993.

[SHL75] Nan C. Shu, Barron C. Housel, and Vincent Y. Lum. CONVERT: A high level translation definition language for data conversion. *Communications of the ACM*, 18(10):557–567, October 1975.

[Sjø93a] D. Sjøberg. Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44, January 1993.

[Sjø93b] D. I. K. Sjøberg. *Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems*. PhD thesis, University of Glasgow, July 1993.

[ST82] Ben Shneiderman and Glenn Thomas. An architecture for automatic relational database system conversion. *ACM Transactions on Database Systems*, 7(2):235–257, June 1982.

[SZ86] Andrea H. Skarra and Stanley B. Zdonik. The management of changing types in an object-oriented database. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 483–495, September 1986.

[TC93] Peri Tarr and Lori A. Clarke. Pleiades: An object management system for software engineering environments. In *Proceedings of ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering*, Los Angeles, CA, December 1993.

[WWFT88] Jack C. Wileden, Alexander L. Wolf, Charles D. Fisher, and Peri L. Tarr. PGRAPHITE: An experiment in persistent typed object management. In *Proceedings 3rd Software Development Environments Conference*, pages 130–142, December 1988.