Schema Evolution

Melliyal Annamalai

Sept. 17, 1996

What is Schema Evolution ?

Managing the problem of type evolution (or class modification) in an object-oriented database environment.

What is Affected ?

- Objects of that type
- Programs that use objects of that type

We examine three approaches:

1. Based on ENCORE: Dealing with both object changes and program behavior changes using versions

2. Based on GemStone: Dealing with mostly object changes by explicitly specifying procedures for each possible change

3. Based on ORION: Dealing with mostly object changes by automatically generating a *correction program (OTGen)* to implement the changes based on declarative tables

Invariant Conditions: Features that have to be preserved across type evolution

- Representation invariant: Representation of an object (storage format, size, variables and constraints) must be as specified by object's class.

- Unique name invariant: Each defined or inherited instance variable and class must have a unique name

- Class hierarchy invariant: Classes and subclasses form a tree in GemStone, (because multiple inheritance is not allowed). Class and subclasses form a lattice in ORION and ENCORE

- Full inheritance invariant: Instances are all inherited by all its subclasses.

- Class kind invariant: Constraint on an inherited variable must consistent with constraint in the superclass

- Dangling reference invariant: There should be no dangling references.

- Typed variable invariant: The type of each instance variable must have a corresponding class

Changes that have to be supported

- Adding/deleting/renaming an instance variable
- Changing the type of an instance variable
- Adding/deleting a superclass from a class's superclass list
- Adding/deleting/renaming a class (type)
- Modifying the constraint on a class (type)
- Making a class indexable (GemStone)

Approaches to Changing Objects

Coercion: Objects are coerced into a *new* version

convert(new_type: Types, instance: Entities) (ENCORE)

There are two approaches depending on *when* its done

- Changes all objects as soon as the type/class changes. Large amount of time is spent during modification. GemStone and ORION follow this.

- Changes objects only when they are used - there is a screening mechanism that checks each object just before it is used. Execution speed is compromised. ENCORE follows this.

Justification in ENCORE

- Use of coercion is not extensible to changing programs
- Constraints can be strengthened, relaxed
- Programs can encounter unexpected interfaces
- Reader's problem: reading values which are not expected (constraint based problem)
- Writer's problem: constraint based problem and instance change based problem

Approach in ENCORE: Use of Versions

Versions are used to deal with programs that use objects whose type has changed.

- A version set is an ordered collection of all incarnations of a particular object

- A version is initialized with the first definition of a type. Each modification results in a new version

- One member of each set is distinguished as the *current version*

- An object is bound throughout its life to a particular version. Can be converted to another version using coercion

- When a type is instantiated, a version can be chosen or the default current version used

Version Set Interface

- An abstraction defined by a type over time
- Its the union of all the type's versions
- A standard interface for all instances of a type, regardless of their version
- Errors might result because of differences between the version set interface and the interface defined by a single version

Handlers solve this problem.

There are two version set interfaces:

1. Union interface

2. Intersection interface

There is a handler for all values that are defined in the version set but are not in the intersection interface (that is, not defined everywhere).

Error conditions and handlers:

- Attempt to write an item which is outside a defined domain: invalid

- Attempt to write an item which is an undefined property or operation: undefined

Handlers are defined for connectors which are not fixed.

Error handlers are inherited by subtypes like other properties and operations.

Approach in GemStone

They list class-modifying operations:

- Rename a named instance variable:
  - Not allowed if new name is already defined in class or inherited from a superclass (maintains representation invariant)
  - Renaming is propagated to all subclasses (maintains full inheritance invariant)
- Making a class indexable:
  - Not allowed if a subclass is already indexable and the constraint on the indexable component would cause a violation of the class kind constraint (maintains class kind invariant)

- Adding a name instance variable:
  - Similar to renaming.
  - A value of **nil** is assigned to the new instance variable
- Making an indexable class non-indexable:
  - All instances of the class have their indexable component removed.
  - This can be propagated to subclasses if the user wishes
- Removing a named instance variable:
  - Not allowed if it is an inherited variable (to preserve full inheritance invariant).
  - This can be propagated to subclasses

if the user wishes

- Modifying the constraint on a variable. Can be:
  - Specialized
  - Generalized
- Removing a class
- Adding a class

Other Considerations in the GemStone Approach

- Authorization: Performing class-modifying operations will become difficult when individual instances are owned by different users

- Concurrency: Locking mechanisms are required to ensure that users do not use the old version when a new version is being committed

- Enumerating all instances in a class: Useful to bring a database to a consistent state immediately after a modification

- Grouping modifications

- Binding: Late binding will attempt to adapt to the new environment. Early bindings will reflect the old environment and may not be correct

Advantages over the filter approach

- Filtering is for a lifetime
- Every access to every object is filtered
- Explicit deletion is expensive

Approach in ORION

- Automatic support for *database reorganization*

- Describes the development of an program generator based on delta of the definitions (difference between the old and new definitions).

- The generated program will automatically convert the old objects to correspond to the new schema

OTGen defines a set of default transformations

OTGen brings to attention:

- Naming conflicts
- Disconnection of lattice
- Type incompatibility
- Violation of typed variable constraints

Temporary violations are allowed

The transformation should maintain:

- Completeness: When a modification is made **all** objects should be transformed to the new definition before they are manipulated further

- Correctness: After modification, each object must correspond to the definition of a class in the new set of definitions.
  - This is maintained since newly created objects and assigned values are checked for type correctness.
  - OTGen assists in maintaining correctness by providing a template of variables for a new object. Table is not complete until each variable transformation is specified.

- Sharing: If two instance variables point to a same object, and
  - Neither instance variable is deleted
  - Neither instance variable's default transformation is overridden
  - Modified object is type correct for each instance variable

  then they will both point to the same object after modification.
  This is achieved by assigning unique UIDs with each object

Transformation Tables

- One entry for each class

- Reflects changes based on default transformations

Default mapping operations performed by OT-Gen:

- Creation of an object in the new database with the same class as the object in the old database

- Replacement of an object in the old database with VOID in the new database when the old object's class is deleted

- Recursive transformation of an old instance variable's value into a value for the same instance variable in the new database

- Initialization of an instance variable in the new database to VOID

The database administrator can override the default tables using the following additional tables:

- Initialization of variables
- Context-dependent changes
- Movement of information from one object to another
- Creation of new objects
- Sharing of information among objects

Example of a transformation table:

Operation: Adding a new instance variable $v3$ to a class C

```
Class C:
   new self: C
      v1: Transform v1
      v2: Transform v2
      v3: void
```

The assignment of value ‘void’ can be overridden by the database administrator:

```
Class C:
   new self: C
      v1: Transform v1
      v2: Transform v2
      v3: 0
```

Example of moving information

Default transformations

```
Class Outer:
    new self:  Outer
        o1: Transform o1
        o2: Transform o2
        i2: void

Class Inner
    new self:  Inner
        i1: Transform i1
```

But the value of Inner's $i2$ is lost

Should be overridden as:

```
class Outer:
    new self:  Outer
        o1: Transform o1
        o2: Transform o2
        i2: Transform o2.i2
```