MICROSOFT REPOSITORY VERSION 2 AND THE OPEN INFORMATION MODEL

PHILIP A. BERNSTEIN, THOMAS BERGSTRAESSER, JASON CARLSON,
SHANKAR PAL, PAUL SANDERS, DAVID SHUTT

Microsoft Corporation One Microsoft Way, Redmond, WA 98052-6399 U.S.A.

Abstract[1] **—** Microsoft Repository is an object-oriented meta-data management facility that ships in Microsoft Visual Studio and Microsoft SQL Server. It includes two main components:

- A repository engine that implements a set of object-oriented interfaces on top of a SQL database system. A developer can use these interfaces to define information models (i.e., schemas) and manipulate instances of the models.
- The Open Information Model, which is a set of information models that cover object modeling, database modeling, and component reuse.

The repository system is designed to meet the persistent storage needs of software tools. Its main technical goals are:

- Compatibility with Microsoft's Component Object Model (COM) architecture
- Extensibility by customers and independent software vendors, so they can add behavior to objects stored by the repository engine and extend information models provided by Microsoft and others.
- Flexible and efficient versioning, configuration management, and checkout/checkin to support team-oriented activities.

This paper describes the programming interface and implementation of the repository engine and the Open Information Model.

Key words: repository, information model, versions, object-oriented database

1. INTRODUCTION

Microsoft Repository is a persistent object manager and object model designed to help users manage descriptions of things, that is, meta-data. It is being used in Microsoft Visual Studio to manage reusable components and exchange object models, and in Microsoft SQL Server to manage database schemas and transformations used for data warehousing. It is a general purpose technology that is being applied by third parties to a variety of other applications, such as asset management, model management, and computer-aided design (CAD). The demand for such technology is growing rapidly, driven in part by multi-tier application environments [23] accessible via the World-Wide Web [25].

The first release of Microsoft Repository was in early 1997 in Visual Basic 5.0, and over a million copies have shipped. The second release, which includes version and workspace management and a much expanded information model, shipped in Visual Studio 6.0 and SQL Server 7.0 in late 1998.

The repository system is composed of two major components. The first is the repository engine, which implements a set of object-oriented interfaces that a developer can use to define information models and manipulate instances of those models. (*Information model* is repository terminology for database schema, often an object-oriented schema that includes behavior.) The repository engine sits on top of either Microsoft SQL Server[2] or Microsoft Jet (the database system in Microsoft Access) and supports both navigational access via the object-oriented interfaces and SQL access to views of the underlying store. The second component is the Open Information Model, which is a growing set of information models that currently covers object modeling, database modeling, and component reuse. These areas are based on the Unified Modeling Language (an OMG standard), the ANSI SQL standard, and Microsoft's Component Object Model, respectively. The repository system also includes a model development kit for extending the Open Information Model and developing new models.

The main technical goals of the repository engine are:

1. Object Model Compatibility – The repository's object model should fit naturally into Microsoft's existing object architecture, called the Component Object Model (COM) [20]. Thus, the repository system should use existing

[1] This paper is a draft, whose final version appeared in I*nformation Systems 22, 4 (1999)*, published by Pergamon Press. It subsumes the paper "Microsoft Repository," by P. Bernstein, B. Harry, P. Sanders, D. Shutt, and J. Zander, published in the *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997, pp. 3-12. Portions of that paper are reprinted here by permission of the Very Large Data Base Endowment.

[2] ActiveX, Microsoft, Microsoft SQL Server, Visual Basic, and Windows are trademarks of Microsoft Corporation.

COM interfaces and implementation technology wherever possible, expose its content as COM objects, and minimize the number of new concepts that the large community of COM users needs to learn.

2. Extensibility – In order to differentiate their tools and add value, it is important that customers and third-party vendors be able to tailor the repository to their needs, both by providing methods on objects stored by the repository engine and by extending persistent state. The latter can be done declaratively, with no procedural logic.

3. Version and Configuration Management – Since meta-data is used in a design environment and shared by teams, it must be versioned, grouped into configurations, and managed using checkout-checkin. The versioning model must be efficient in time and space, functionally rich, and flexible — easy-to-use by customers who use it unmodified and easy-to-customize by others.

This paper explains how the repository engine attains these goals. Fitting an object-oriented database system (OODB) or repository engine into an existing object model is a delicate activity, which we explain in detail (goal 1). The result is rather different than a C++ or Smalltalk based OODB, in part because COM is a binary standard, not a language API, and is more strongly interface-based than class-based. Since COM has powerful extensibility features, integrating the repository with COM produces an extensible engine without requiring many new extensibility features of its own (goal 2). And while versioning techniques are well known, not many products or prototypes support all of the above requirements for versioning fine-grained objects linked by arbitrary relationships (goal 3).

The paper explains how the underlying COM model affects the structure and use of the Open Information Model, which knits together several large existing standards: UML, SQL, and COM. The use of COM and the development of OIM drove requirements for our modeling environment, which is also described here.

This case study is useful not only for its description of repository-related technologies, but also because so little has appeared in the research literature about complete repository systems. A general introduction can be found [3]. The PCTE standard is described in [24]. Many OODBs are used as repositories, though they are not quite the same, as explained in [2]. Still, the large literature on OODB systems is relevant to the present work, such as [5], [8], [9], [13], and [26], as is the literature on versioning, such as [7], [14], and [22]. Another popular approach is to circumvent the problem of integrated meta-data management by exchanging meta-data between independent systems, using standard protocols, such as CDIF [6] or XML [25]. A comparison of our design with these and other systems and approaches is beyond the scope of this paper.

The paper is organized as follows: Section 2 presents background on COM necessary for understanding the object model of Microsoft Repository. The data definition facilities of the object model are described in Section 3. The repository engine's data manipulation functions are described in Sections 4-6: Section 4 describes object and relationship manipulation; Section 5 describes transactions and caching; and Section 6 describes version and workspace management. The mapping of the repository's object model onto a SQL database is presented in Section 7. Section 8 describes our goals and approach to information modeling, and Section 9 summarizes the content of the Open Information Model. Section 10 is the conclusion.

2. COM AND AUTOMATION

2.1 The Component Object Model

Microsoft's Component Object Model (COM) is the foundation of Microsoft's object architecture. It is a binary standard that describes component-to-component early-bound calling conventions in a language-neutral fashion, so that components written in different languages can seamlessly call one another.

In COM, a *class* is an executable program image. It can implement multiple *interfaces*, where each interface is a set of methods, called *members*. All of the repository engine's interfaces are COM interfaces.

Each class has a class identifier (class ID), which is a 128-bit globally unique identifier (GUID). Given a class ID, the function `CoCreateInstance` creates an *object*, which is an instance of the class. `CoCreateInstance` finds the class's executable by looking up the class ID in the system registry, which is a small hierarchical persistent store managed by Windows operating systems. Entering the class ID in the registry is part of the class's installation procedure.

An interface's specification includes an *interface identifier* (IID), which is a GUID, and an ordered list of its method names, together with each method's parameters. After an interface is published, its specification is immutable. Therefore, to enhance a published interface, one must implement a new interface with a new IID.

By convention, interface names begin with the letter *I*. Figure 1 gives a graphical representation of a class with multiple interfaces; interfaces are depicted as "lollipops" attached to the class or instance of the class. Methods are ordinarily not shown in this representation; for example, the `IForm` interface could support the `Resize` and `AddControl` methods, which are not shown in the figure.

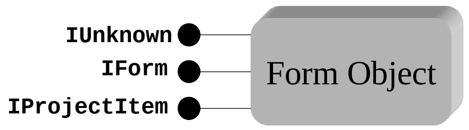


Figure 1 Representation of a COM Component

A COM interface, IX, can directly inherit from at most one other interface, say IY, in which case IX includes the members of IY, in addition to those explicitly defined on IX. This is type inheritance, not implementation inheritance; that is, the definition of IX inherits the definition of IY. Every COM interface inherits, directly or indirectly, from `IUnknown`. COM supports a form of multiple inheritance, in that a class can implement many interfaces. It also supports polymorphism, in that an interface can be implemented by many classes.

Every COM class, and hence every instance of that class, implements the interface `IUnknown`. `QueryInterface` is a method on `IUnknown` that allows a client to ask a COM object if it supports a particular interface, given the interface's unique IID. If the object supports the interface, it returns a pointer to that interface on that object. In this case, the client knows exactly what behavior the object will provide, because interface definitions are immutable. If it does not support the interface, it returns null. Thus, in Figure 1, an instance of `Form` would respond positively to a call on `QueryInterface` given `IForm's` IID and would therefore have to support all of the methods specified for `IForm`.

The `QueryInterface` mechanism helps cope with type evolution, as follows:

- Since interface definitions are immutable, to change the behavior of an interface, one must define a new interface. Over time, one may have several different interfaces that are effectively versions of each other. We use "version" loosely here; each "version" is an independent interface insofar as COM is concerned.
- A class can support several different interfaces, which may be different versions of an interface that has evolved over time.
- A client can cope with multiple versions of a class's interface as follows: The client queries for the interface version it prefers. The class's instance replies yes or no. If it answers no, the client queries for its second favorite version of the interface, and so on. The client and object can interoperate if the client finds an interface that it knows how to use and that the object supports.

This mechanism allows classes and their clients to be independently upgraded.

Every class has a *class factory*, which can create instances of the class. The class factory returns a pointer to an interface on the object. After receiving this pointer, a client can call methods on that interface of the object (locally, or remotely using Distributed COM (DCOM)), or can use `QueryInterface` to find other interfaces on the object and call methods on them.

A COM class, C, can be extended by wrapping it. The wrapper object creates a subsidiary inner object which is an instance of the original class that it wants to extend. The outer object controls the lifetime of the inner object, whose existence is hidden from others. This technique is called *aggregation* if the wrapper exposes any of C's interfaces directly; otherwise it is called *containment*.

The technique involves writing a class, C_1, that supports the extended behavior, which may consist of new interfaces and/or wrapped implementations of C's interfaces. C_1 only needs access to C's executable (not its source code). To make this work, C_1 replies to `QueryInterface` only on interfaces that C_1 implements (i.e., those it wrapped or its new interfaces). It delegates calls on `QueryInterface` for other interfaces to C. To be aggregable, C must follow a certain discipline in its implementation of `IUnknown`. These and related details of COM aggregation are explained thoroughly in COM documentation and are well known to COM developers [20].

A class can aggregate many classes and be aggregated by many classes; in this sense, COM supports multiple inheritance of implementations. As we will see later, the repository's support of user-defined methods and much of its extensibility is a direct application of COM aggregation.

A COM interface for a class is implemented in memory as a *vtable* (i.e., virtual function table). Each object of that class has a pointer to that vtable plus some object-local data structures. A vtable has an entry for each of its members, which points to the in-memory executable for that member. The pointer returned by `QueryInterface` points to that object's pointer to that interface's vtable. In this sense COM is a binary calling standard. Since the caller is assumed to know the order (and meaning) of member entries in the vtable, COM is best suited for early-bound access.

2.2 Automation

Automation is a mechanism for late-bound calling of objects, originally developed for Visual Basic and later integrated with COM. Automation functionality is captured by the COM interface *IDispatch*. `IDispatch` supports a method, `Invoke(M, [parm1, parm2, …])`[3], which implements a late-bound call to method M with parameters parm1, parm2, …, on the object (i.e., the one that implements `IDispatch`). For an interface on an object to be invoked in this way, it must inherit from `IDispatch`, in which case it is called a *dispatch interface*.

A dispatch interface can have many members, each identified by a *dispatch ID*, which is the value used for parameter M in `Invoke. IDispatch` also includes a method `GetIDsOfNames`, which maps member names to dispatch ID's (for efficient late-bound access), using information contained in a type library (described below).

A member of a dispatch interface can either be an ordinary method or a *property*, which is a shorthand for saying it has methods `get_Foo` and `put_Foo` for the property Foo. A property can either be scalar-valued or object-valued. One important and popular type of property value is the collection object, which supports the following methods:

- `Add` – inserts an element
- `Count` – returns the cardinality of the collection
- `Item` – retrieves an element by index or key
- `Remove` – deletes an element identified by index or key, and
- `_NewEnum` – returns an enumerator (i.e., cursor) on the collection, which can be traversed by calls to the `Next` method.

An interface can be both a COM interface and a dispatch interface, called a *dual* interface. This is an optimization that allows some members of a dispatch interface to be called through the early-bound COM mechanism. A caller who knows the definition of the interface at compile time can use this information to make an early-bound call and therefore avoid the overhead of interpretation by `IDispatch`. Most interfaces to the repository engine are dual interfaces.

The "standard" implementation of `IDispatch` (i.e., for Visual Basic) uses a *type library* object to look up the definition of external interfaces it is asked to invoke. To produce a type library, a class developer writes an interface definition in Microsoft's interface definition language (MIDL) and compiles it into a type library, which can either be stored as part of the class's executable or in a separate file. Type libraries can be directly accessed via their own interfaces, such as `ITypeLib` and `ITypeInfo`. Often, they are accessed indirectly via `IDispatch::GetIDsOfNames` (a member `M` of interface `IX` is denoted `IX::M`).

Visual Basic syntax translates directly into calls on `IDispatch`. For example, in this program fragment:

```
DIM X as Object
X.Foo = 7
```

the Visual Basic implementation (called an *Automation Controller*) uses `GetIDsOfNames` to look up Foo, and then uses `Invoke` to call `put_Foo`(7). If `X` supports multiple interfaces, then the above program accesses property `Foo` on its *default* interface. Another interface, `IBar`, could be accessed like this:

[3] `Invoke`'s signature is more complex, but the details are unimportant for this discussion.

```
DIM Y as IBar
Set Y = X
Z = Y.MyFunction()
```

The statement "`Set Y = X`" calls `QueryInterface` on `X` for `IBar` and assigns that value to `Y`.

3 OBJECTS AND TYPES

3.1 The Repository's Object Model

COM and Automation are used as the native object model by the vast majority of programming tools for Microsoft operating systems. It was therefore a requirement that the repository engine's functionality be exposed as a set of COM and Automation objects. These objects are in-memory representations of the information held in the repository database and support a set of repository-specific dual interfaces. An object is a repository object if and only if it supports a certain set of these repository-specific interfaces.

The repository engine supports four main kinds of objects:

- **Repository Session** - represents the repository database itself. It behaves much like a database session.
- **Repository Object** - represents the persistent state of an object in a repository. That state consists of the object's properties and collections.
- **Relationship Object** - represents a connection between two repository objects. A relationship can have properties (unlike the ODMG standard [4], where a relationship is not an object and has no attributes). The relationship's connection and properties are stored in the repository database.
- **Collection Object** - represents a set of relationship objects. A collection of relationships is accessed and updated using the standard collection methods: `Add`, `Count`, `Remove`, `Item`, and `_NewEnum`.

The repository engine is a type-driven interpreter. A user specifies an information model that consists of a set of type definitions, namely, class, interface, property, method relationship, and collection definitions. The repository engine provides methods for creating objects that are instances of these classes, and for storing and retrieving these objects' properties and relationships to and from the repository database. One good way to understand the repository's capabilities is to understand what can be expressed in type definitions.

3.2 The Type Model

Repository type definitions are ordinary repository objects that have certain type-specific properties and relationships that are interpreted by the repository engine. For example, a class definition is an object that has a property containing its unique identifier and a relationship to the interfaces it implements. This usage of its own storage mechanism for type definitions is analogous to a SQL engine, which stores type definitions as rows of tables (i.e., in its catalog).

Type definitions are grouped into repository type libraries. These have the same logical structure and namespace behavior as Automation type libraries. Having the same namespace behavior is important so the repository can match Automation semantics for name-based access to properties and collections.[*] Specifically, class and interface names must be distinct (i.e., in `DIM X as ABC`, `ABC` could be a class or interface), and member names must be unique relative to an interface (i.e., in `Object.MyMember`, `MyMember` could be a method, collection, or property). Classes and interfaces have unique class ID's, as in COM.[*] The type library abstraction also provides a mechanism for partitioning large information models. That is, each type library can be used for type definitions relating to a specific subject area, such as component descriptions, SQL schemas, or OLAP schemas.

A type library contains definitions of the following kinds of objects:

i. Class – defines the class ID, class name, and which interfaces it supports, one of which is its default interface (for Automation).[*]

[*] To highlight the effect of COM and Automation on the repository design, we tag each sentence that describes such an effect by an asterisk.

ii. Relationship class – defines which collections (on which interfaces) are connected by instances of the relationship class.

iii. Interface– defines the interface ID, which properties, collections, and methods are members of this interface, and which interface it inherits from.*

iv. Property– defines properties of the property, such as its data type and its mapping to an underlying SQL column.

v. Collection– defines properties of the collection, such as min and max cardinality. These are properties of endpoints of a relationship type, called *roles* in some object models.

vi. Method– defines properties of the method, such as its dispatch ID.*

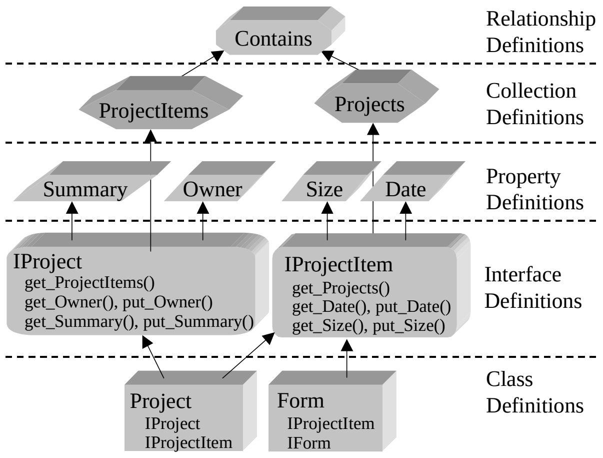


Figure 2 An Information Model.

Interfaces are defined on classes, and properties and collections (i.e., relationships) are defined on interfaces.* Figure 2 shows example definitions for a contrived information model, where each polygon represents a data definition object. Interface `IProject` describes project containers and `IProjectItem` describes objects that can be put into project containers. The Project class supports both `IProject` and `IProjectItem` (since a project can have subprojects), while the Form class supports `IProjectItem` but not `IProject`. Properties and relationships that are specific to forms are captured by `IForm` (shown in the Form class but not defined in Fig. 2). The relationship Contains is accessible via the `ProjectItems` collection on `IProject` and the `Projects` collection on `IProjectItem`.

An instance of the Project class is shown in Fig. 3. In the figure, MyProject is an instance of the Project class and therefore supports `IProject` and `IProjectItem`.

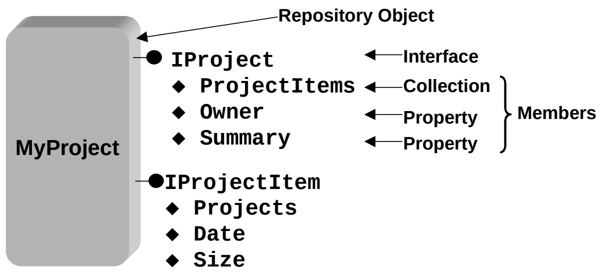


Figure 3 A Repository Object

Type definitions are repository objects. Like all repository objects, they are instances of classes, which in turn support interfaces that have properties and relationships stored in the repository. For example, the definitions of `IProject` and `IProjectItem` are instances of the interface definition class, InterfaceDef, which supports the interfaces `IInterfaceDef` (which provides the behavior unique to interface definitions) and `IReposTypeInfo` (which allows interfaces to be the target of `DIM` statements in Visual Basic*). Thus, the information summarized in (i) – (vi) above is captured by the interfaces and relationships summarized in Fig. 4.

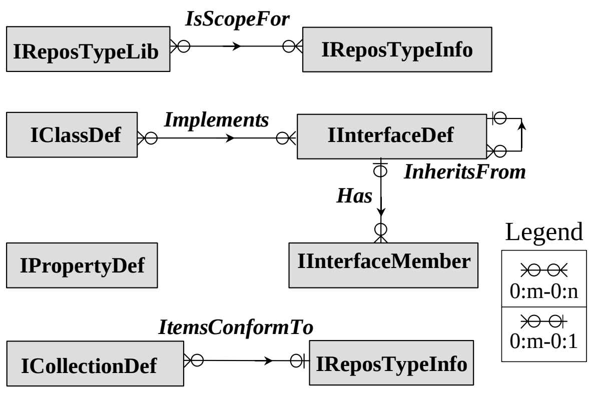


Figure 4 Repository Type Model

The classes that use these interfaces are as follows:

i. ClassDef – supports `IReposTypeInfo` and `IClassDef`

ii. RelationshipDef – supports `IReposTypeInfo` and `IClassDef` (relationship-specific information is in `ICollectionDef,` so no "`IRelationshipDef`" interface is needed)

iii. InterfaceDef – supports `IReposTypeInfo` and `IInterfaceDef`

iv. PropertyDef – supports `IInterfaceMember` and `IPropertyDef`

v. CollectionDef – supports `IInterfaceMember` and `ICollectionDef`

vi. MethodDef – supports `IInterfaceMember`

All type definitions in an information model are represented as COM objects that are instances of the above classes.

The Repository Type Model classes are described as instances of themselves. That is, there is an instance of ClassDef for each of the above classes: ClassDef, RelationshipDef, etc. There is an instance of RelationshipDef for each of the relationships in Fig. 4: IsScopeFor, Implements, Has, etc. There is an instance of InterfaceDef for each of the interfaces in Fig. 4: `IReposTypeLib`, `IClassDef`, `IInterfaceDef`, etc. And so on. In this sense, the repository is self-describing. This characteristic is useful for model-driven tools, such as generic browsers and scripting languages, which need to discover the information model at run-time and which should be able to view the repository's type model in the same way as models customized for applications. It also positions the repository to apply its own features to its own type definitions, now and in the future. For example, Repository Type Model classes are extensible by customers. Another example is that, when the repository engine supports versioning of type definitions, the operations to manipulate those versions will be identical to operations for manipulating versions of other kinds of objects.

Two aspects of interface definitions are worth noting:

- The repository engine supports interface inheritance with the same semantics as COM.[*] That is, if an interface I2 InheritsFrom an interface I1, then all of the properties and collections that are defined on I1 are also available on I2.
- An interface can include custom methods, whose existence can be documented in the interface definition stored in the repository. The information model developer is responsible for implementing such methods (see Section 8.2).

The repository type model will track future evolution of COM, for example, by incorporating the enhancements that are offered in COM+ [15].

4. OBJECT MANIPULATION

4.1 Objects and Versions

Conceptually speaking, a repository contains a set of repository objects, each of which consists of a set of *repository object versions* that are connected by successor-predecessor relationships. This statement is somewhat vague, in that it does not identify which of these entities are COM objects and how those COM objects relate to persistent state. Speaking more concretely, each version of a repository object is a COM object, called a *repository object version*, which represents some persistent state. That state has two parts: some version-specific state; and some version-independent state of the version's repository object, which is the same for all versions of the repository object.

By design, the version-specific and version-independent behavior of a repository object version are orthogonal. Thus, the behavior of a repository object can be used and fully understood without any knowledge of its version dimension. In fact, the first release of the product did not support versioning behavior at all. The second release added versioning by introducing an orthogonal set of interfaces, i.e., without affecting the syntax or semantics of the release one interfaces. We exploit that orthogonality here by first explaining the non-versioned behavior of repository objects in the remainder of Section 4 and discussing versions and related functionality in Section 6.

4.2 Repository Sessions

To use the repository, one starts by creating a *repository session*, which is an instance of the class Repository. One can then call the repository session's `Create` method to create a new repository database, which creates tables required by the repository engine in a given SQL database. Or, one can call the repository session's `Open` method to open an existing repository database, that is, a SQL database that had previously been created as a repository.

After executing a `Create` or `Open` method, the repository session is bound to a repository database, so one can start creating and accessing objects. The `CreateObject` method on the repository session creates a new repository object of a given class. One can get existing repository objects using the following techniques:

1. The `Create` and `Open` methods return the repository's unique root object. By convention, the root is connected directly or indirectly to all other repository objects in the database.
2. The `ObjectInstances` method on a class or interface definition returns a collection of all repository objects that are instances of that class or support that interface, respectively. Usually, class and interface definitions are well known (e.g., are defined in header files), since their object ID's are the same in every repository.
3. Get a set of objects that satisfy a query
4. Get the object that has a given object ID
5. Follow a relationship from an object previously obtained by one of these five techniques.

Techniques (3) – (5) are described later in Section 4.

4.3 Repository Objects

Object Identifiers

Each repository object has a globally unique 20-byte opaque *external ID*. It can be automatically assigned by the repository or supplied by the caller to `CreateObject`. The latter is useful to give an object and its replica the same identity (e.g., a type definition that's stored in many repositories). Since external ID's are globally unique, they can be made well known. Therefore, the repository session supports a method `get_Object`, which loads an object given its external object ID.

Repository objects also have an *internal ID* that is an 8-byte compressed representation of the external identifier, an important storage optimization. The internal identifier is always assigned by the repository engine, and a given object can have different internal identifiers in different repositories. Object identity can be determined by comparing external object ID's. If the objects are known to be from the same repository, then the internal ID's can be used instead, resulting in a cheaper comparison.

Every repository object supports the interface `IRepositoryObject`, which includes most of the generic behavior of repository objects. This interface includes methods get the object's external or internal ID. Other methods on this interface are described later.

Properties

Repository objects can have single-valued scalar properties, which are accessible using `IDispatch` (for Automation) and generic `get_value` and `put_value` methods (for COM). The former allows properties to be accessed using ordinary Visual Basic syntax*, such as

```
DIM X as RepositoryObject
X.Foo = 7
```

where Foo is a property of X's default interface.

Depending on the development tool, early-bound model-specific COM access is also available. For example, one can use the Active Template Library (ATL) of Microsoft Visual C++ to create early-bound wrappers for dispatch interfaces [17].

4.4 Relationship Objects

A relationship is bi-directional. That is, it can be followed from either of the repository objects it connects. Like a repository object, it can have properties. Unlike a repository object, it cannot have relationships or methods, though the latter restriction is likely to disappear in a future release.

Selecting the appropriate amount of semantics for relationships is a hard decision when designing a persistent object model. Inevitably, some customers want high-functionality relationships, for example, n-ary relationships and relationships between relationships. However, everyone wants high-speed relationships, which argues for limited relationship functionality, since advanced relationship functions are usually costly. Our high-end customer scenarios required the support of attributed relationships, since it is more efficient than turning the relationship into a first-class repository object. The repository engine supports this feature, which we found attractive because it adds storage and run-time expense only to those information models that use it. Had we gone much further in adding relationship functionality, we would probably have had to introduce a lightweight relationship type that is simply an object reference, as in the ODMG model. Depending on customer needs, we may follow this path in the future.

Each relationship is an instance of a *relationship class*. A relationship class definition connects two collection definitions (on the same or different interfaces), called the *origin* and *destination*. Although a relationship instance can be traversed in either direction, some semantics of the relationship is sensitive to the relationship's polarity indicated by origin and destination. More on this later in the section.

Starting from a repository object, one can access a relationship collection, access a relationship within the relationship collection, and then access the repository object on the other side of the relationship. The repository object where the traversal starts is called the *source* and the one where traversal ends is called the *target*. That is, the concepts of source and target are relative to the traversal direction. So, the source could be on the origin or destination side of the relationship's relationship class. Notice that a relationship is actually a member of two collections, one on its source and one on its target.

Relationship collections are accessed using `IRelationshipCol` ("`Col`" abbreviates "Collection"), which supports the standard collection methods. In the common case where a relationship's properties are not present or needed, one can skip over the relationship object and go directly from source to target, by using methods on `ITargetObjectCol` instead of `IRelationshipCol`, both of which are supported by the collection class. That is, the same collection can be viewed as either a collection of relationships (via `IRelationshipCol`) or a collection of repository objects (via `ITargetObjectCol`) that are targets of those relationships. This ability to skip over relationship objects avoids one disadvantage of attributed relationships — that it makes programs that don't need such attributes more verbose.

For example, consider the Contains relationship between `IProject` and `IProjectItem` in Fig. 2. Contains relationships would be accessed via the relationship collection ProjectItems on interface `IProject` on Project objects and via the relationship collection Projects on interface `IProjectItem` on Form objects. Figure 5 is an instance-level view of this model, showing COM objects. The ProjectItems collection for the instance of the Project labeled MyProject has three relationships, one of which, labeled x, points to the instance of Form labeled MyForm. MyForm, in turn, supports `IProjectItem` and therefore has the collection Projects, which contains two relationships each of which points to an instance of Project, one of which is x pointing to MyProject.

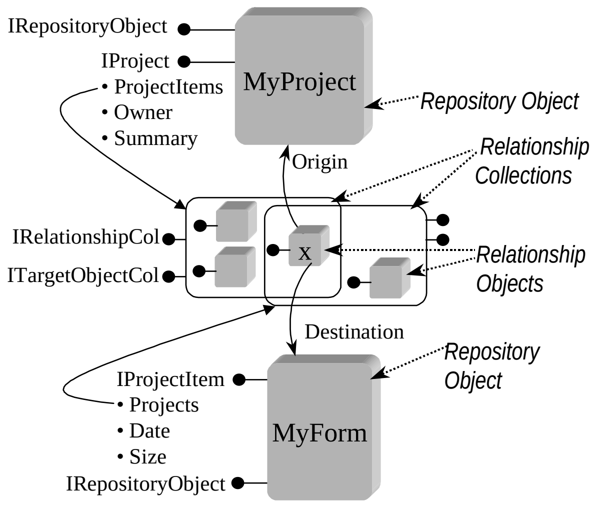


Figure 5 Relationships and Relationship Collections

Using `IRelationshipCol` on the `ProjectItems` collection, one can access each relationship, each of which points to a repository object supporting `IProjectItem`. Or, using `ITargetObjectCol` on that collection, one can skip over the relationship objects, thereby viewing the collection as a set of repository objects supporting `IProjectItem` (not shown in Fig. 5).

Much of the interesting semantics of a repository is captured in the behavior of relationships. In ours, a relationship class can have three kinds of semantics: naming, sequencing, and delete propagation.

A relationship can have a name, which identifies the destination object relative to its origin. The origin collection definition of the relationship class specifies whether it is a naming relationship and, if so, whether names are case sensitive and/or unique (i.e., whether two instances of the relationship from the same origin must have different names). The relationships in an origin collection of a naming relationship are sorted by name.

By assigning names to relationships rather than to objects, a repository object can have different names in different contexts, since it can be the destination of more than one relationship. For example, if the Contains relationship type in Fig. 2 is a naming relationship, a form could have different names in different projects, as shown in Fig. 6, where a form is named MyForm in Project1 and YourForm in Project2.

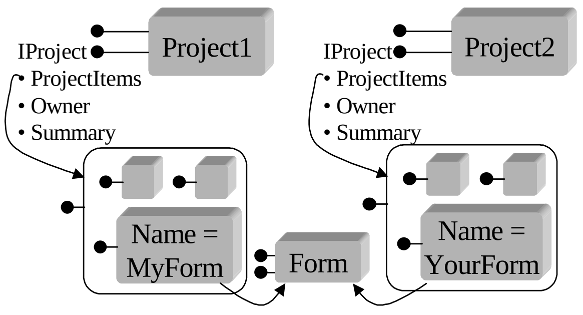


Figure 6 Named Relationships

The flexibility of allowing an object to have different names in different contexts can be inconvenient when an object has the same name in all contexts. To avoid having to assign the same name to all relationships to an object, the repository engine implements a special interface, called `INamedObject`, which has one property called `Name`. Model developers specify which classes implement `INamedObject`. If a repository object O (i.e., its class) supports `INamedObject`, then the method `IRepositoryObject::put_Name` assigns the same name to the `Name` property of `INamedObject` on O and to all naming relationships to O. An update to the `Name` property of `INamedObject` updates that property only.

The destination objects within a relationship collection can be sequenced. This is indicated by setting a flag on the origin collection definition. For sequenced collections, the standard collection methods (`Add`, `Remove`, `Item`, `Count`, `_NewEnum`) are augmented with `Insert` and `Move` methods on relationship collections to control the sequencing. If a naming relationship is also sequenced, the collection is ordered by sequence, not name. Sequencing is useful in many design scenarios, such as ordering the column definitions of a table definition and ordering the member definitions of an interface definition.

Ordinarily, deleting a relationship R affects only R. However, it is sometimes desirable that an invocation of `Delete` propagates the delete beyond R. This propagation is controlled by the delete propagation flag on the origin collection definition of R's relationship type. If the flag is set for R's origin collection definition and R is the last relationship of its type that points to R's destination object D, then deleting R causes D to be deleted too. This is useful for containment hierarchies, where an object should be deleted when the last containment relationship to it goes away. In this case, the container side of the containment relationship type should be defined as origin with its delete propagation flag set.

When developing an information model, it is common practice to ensure that one of the incoming relationship types of each class propagates deletes. This avoids having to write application code to do explicit garbage collection.

The propagation of other operations can be quite useful [21]. However, since the algorithms involve transitive closures, they are rather tricky to implement efficiently. Therefore, additional operation propagation capability was deferred to later releases.

4.5 Support for IUnknown

Some of a repository object's interfaces are generic interfaces supported by every repository object (e.g., `IRepositoryObject`, `IDispatch`). Others are custom interfaces defined in the information model. The properties and relationships on these custom interfaces are implemented by the repository engine by interpreting type definitions. Exposing these properties and relationships through Automation involves making them accessible via interfaces that inherit from `IDispatch`.[*]

The repository object class is the core of the repository engine's type interpreter. Creating an instance of a custom class (e.g., by calling `IRepository::CreateObject(clsid)`) causes the creation of an instance of the engine's generic implementation of the repository object class, passing in the class ID of that custom class. The generic repository object class imitates the given custom class by interpreting that class's type definitions.

There is a technical problem in implementing `IUnknown::QueryInterface` for this generic repository object class. Consider an object O that is an instance of the class Project, which implements the interface `IProject`, whose interface ID is `IID_IProject`. A call to `QueryInterface(IID_IProject)` on O must return a pointer to a pointer to a vtable for `IProject` in O (as described at the end of Section 2.1). However, O's implementation is the generic repository object class, which did not and could not know about any custom interfaces when it was compiled. Therefore, the repository engine effectively synthesizes the custom interface at the time `QueryInterface` is called.[*] It does this by creating a C++ object that contains a pointer P to the repository engine's generic implementation of `IDispatch`, along with a bit of local data that identifies the custom interface it implements (in the example, that interface is `IProject`). A pointer to P is returned by a call to `QueryInterface(IID_IProject)`.

4.6 Support for Model-Driven Tools

Model-driven tools need to discover information models at run-time. This can be done by traversing type information stored in the repository. For example, consider a generic browser that displays all of the properties and collections of all of the interfaces defined on each object. For each repository object O, it would proceed as follows:

1. Get the object ID OID_O of the type of O (via `IRepositoryObject::get_Type`).
2. Get the class definition object CDefO of OID_O (via `IRepository::get_Object`).
3. Get the collection ColInt of interfaces implemented by CDefO (via the relationship `IClassDef`-*implements*-`IInterfaceDef`, see Fig. 4).
4. For each interface definition object IntDef in ColInt, do steps 5 – 9.
5. Set Answer = {}

6. Get the collection ColMem of members supported by IntDef (via the relationship `IInterfaceDef`-*has*-`IInterfaceMember`, see Fig. 4).

7. Set Answer = Answer $\cup$ ColMem

8. If IntDef has an inherits-from relationship to another interface definition IntDef1, then set IntDef = IntDef1 and repeat steps 6 and 8.

9. Display each property and collection in Answer

As a convenience, the repository offers a more direct way to get this information. It supports an interface `IRepositoryDispatch`, which inherits from `IDispatch` and supports one method, `Properties`. This method returns a collection of the properties and relationship collections defined on this interface, including those that are inherited from ancestor interfaces. Thus, the `Properties` method does the work of the nested loop in steps 5-8 above. To benefit from this feature, each interface defined in the information model should inherit from `IRepositoryDispatch`, rather than `IDispatch`.

The repository also supports the COM equivalent of `QueryInterface` for Visual Basic programmers.* Recall from the end of Section 2.2 that one can force an execution of `QueryInterface` in Automation by declaring an object variable to be of a particular interface, as in "`DIM Y as IBar`." But this only works for interfaces known to an application at compile time. To give the same capability to model-driven tools, which discover the information model at runtime, repository objects support a method called `Interface`, which takes an interface as a parameter and casts the object to the requested interface. For example, steps 5-8 above could be replaced by the following Visual Basic statement:

```
Set Answer=O.Interface(IntDef.Name).Properties
```

4.7 Queries

For most purposes, the user of the repository (a tool programmer) calls methods on COM objects. However, users sometimes prefer issuing SQL queries to the repository database for faster or more complex retrievals. They can do this using the `ExecuteQuery` method on `IRepositoryODBC`, which is supported by the repository session class. `ExecuteQuery` takes a SQL query that includes internal object ID and optionally class ID in the SELECT clause, and returns a collection of repository objects — one repository object for each object ID returned by the SQL query. That is, it runs the query and casts the returned rows as repository objects. Notice that the repository engine can perform this function by simply passing through the query to the underlying database system without parsing or otherwise examining the SQL.

An application should never update the repository's SQL tables directly using SQL statements, since the repository engine's update methods maintain the integrity of the database in subtle ways that a user could easily miss.

We describe the physical database schema in Section 7. Of course, for data independence, users should define relational views of that physical schema and pose their queries on these views.

5 TRANSACTIONS

Advanced transaction capability was not a goal of our first two releases. Rather, we wanted to minimize the implementation effort by passing through the transaction behavior of the underlying SQL DBMS. Still, even this modest goal required that we include some transaction functions in the repository engine itself.

5.1 Programming Model

Like most database execution models (e.g. ODBC [11]), we attach transaction behavior to the user's connection to the database, which in our case is a repository session. Thus, each repository session offers the `Begin`, `Commit` and `Abort` methods.

Every repository object is loaded in the context of a repository session. All methods on a repository object execute within the transaction of the session that loaded it. Thus, the loaded object must retain that session context. Since session context contains transaction context, the latter need not be passed as a parameter to any calls on the object.

Each repository session allows only one transaction to execute at a time. To have two concurrent transactions on the same repository database, one can create two repository sessions connected to the database.

Transactions are flat, i.e., not nested. The repository engine counts nested calls to `Begin` and `Commit`, and treats all but the outermost `Commit` as a no-op. A call to `Abort` always aborts the currently active transaction (initiated by the outermost `Begin`).

Methods within a transaction read committed data. Therefore, a transaction's updates are isolated from other transactions until it commits, at which time the updates are permanently installed in the database. That is, degree 2 (read committed) consistency is the default [1], [12].

Like other DBMS designers before us, we found that degree 3 consistency was fairly low on our customers' priority list, so we deferred serializability for a later release. However, we do offer a lock primitive that allows users to explicitly synchronize access to shared data and thereby get the effect of two-phase locking, albeit with some application programming.

5.2 Transactions and Caching

Conceptually, each repository session has an associated cache of repository objects and relationship objects that were accessed, and therefore loaded into memory, via that session.

Suppose an application has two repository sessions connected to the same database in the same process. If the application loads a particular repository object, RO, through both sessions, it will get two COM objects representing RO. This is required because each repository (COM) object retains the context of the repository session that loaded it. To avoid a cache coherency problem, the repository engine ensures that both COM objects share the same cached copy of RO's persistent state.

Ordinarily, the repository engine buffers each transaction's updates in the engine's cache. When the transaction issues a Commit, the engine uses the cached updates to generate dynamic SQL and/or stored procedure calls to update the underlying database system. Of course, a long transaction could overflow the engine's cache. A transaction can reduce the probability of overflow by setting the session's *new cache* option. This ensures that the repository session gets its own cache and is the only session that is accessing that repository database. Still, overflow is possible. If it occurs when a transaction is running in new cache mode, then the engine automatically starts generating updates to the SQL database, even though the transaction has not yet committed, and frees up the cache formerly occupied by the updates.

Two repository sessions on the same repository database share the same repository engine cache. Therefore, updates by a transaction T in one repository session are visible to transactions in the other repository session as soon as T commits. Repository sessions in another process, P, will also see T's updates if P loads those updated objects after T commits. If P already loaded those objects before T committed, then P's cache is stale relative to T, so P's clients will not see T's updates until P's repository engine refreshes P's cache, which it does periodically.

Methods are offered to tell the repository engine to refresh its cache immediately, so an application can obtain an up-to-date view of the repository database whenever it needs one. One can explicitly refresh the entire cache or individual objects and collections. This explicit refresh seems rather crude, but actually reflects the behavior of most of the tools that would use the repository. Most Windows-based tools, beginning with the Explorer, offer an explicit refresh.

6 VERSIONS AND WORKSPACES

6.1 Version Model

The main goal of versioning is to enable the reconstruction of old states of an object. So in contrast to Version 1 of Microsoft Repository (V1), where a repository object has a single state, in Version 2 (V2) a repository object can have many versions, each of which represents one of the object's historical states. Versions of an object are related by *successor* relationships, indicating the order in which the states arose. Each version has a globally unique *version ID* and has its own property and collection values. Since type definitions (e.g., class, interface, and relationship definitions) are ordinary repository objects, in principle they could be versioned, but this is not supported in the V2 release.

A version is in the *frozen* or *unfrozen* state. The CreateObject operation to create a new repository object creates the first version of the object, initially unfrozen. The properties and origin collections of an unfrozen version are updatable, but those of a frozen version are not. The Freeze operation freezes a version.

Given a frozen version FV of object O, the CreateVersion method returns a new version NV of O, where NV is a successor of FV. Requiring a version to be frozen before creating a successor allows the repository engine to use *delta storage*, that is, store only values of a version that differ from its predecessor.

Typically, an object is left unfrozen until the developer of the object wants to either save it as a possible state to revert to or share it with others.

The successor relationship between versions induces a directed acyclic graph as follows: Multiple invocations of CreateVersion on the same version, V, cause V to have multiple successors; after the first, each successor starts a new *branch* of the version graph. Later, two versions of an object may be merged using the MergeVersion method. An example is in Fig. 7, where the circled numbers indicate the order of operations. Methods are available to traverse the version graph by getting a version's successors and predecessors.

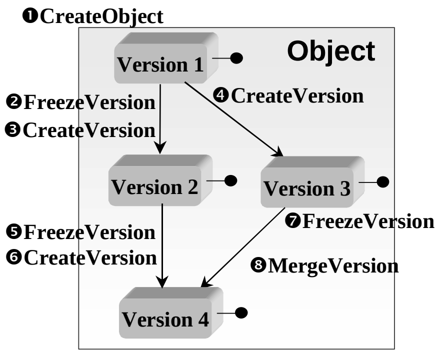


Figure 7 Version Graph

MergeVersion on an unfrozen version V of object O takes two parameters: a frozen version FV of O and a flag that identifies either V or FV as *primary*. It makes FV a predecessor of V and merges the state of FV into V as follows: It finds a least common ancestor of V and FV, called the *basis version*, BV, and compares V and FV to BV. For each property P of O, if only one of V or FV has a different value of P than BV (i.e., it updated P after BV), then the updated value is assigned to P in V. If both V and FV updated P, then the value of the primary is assigned to P in V. The same is true for collections, except that the rule can be applied either to the whole collection or to each relationship within the collection. A flag on each collection's type definition drives this choice. For example, if a collection has maximum cardinality 1, then merging the whole collection would be more appropriate, to avoid creating a collection with cardinality two when merging two collections that each have one member.

Although the above semantics of MergeVersion cover many common cases, some applications may prefer another algorithm for merging state. Therefore, one can override the merge algorithm for a class in a wrapper, using COM aggregation.

6.2 Versions and Relationships

For power and flexibility, it is important to support relationships between individual versions of objects. For ease-of-use and backward compatibility with V1, single-version views are important too. We describe version-to-version relationships here and single-version views in Section 6.4.

In V2, for each relationship type, any version that supports the relationship type's origin interface may be related to any version that supports its destination interface. In the object model, this capability is exposed by having each relationship object connect a source version to a set of target versions. More precisely, each relationship supports a method TargetVersions that returns a collection of all the versions of the target object that are related to the source version. Version-to-version relationships are added and removed by adding and removing items in the TargetVersions collection. For example, Fig. 8 shows a relationship from version 5 of object X to versions 1 and 2 of object Y. To add a version-to-version relationship from version 5 of X to version 3 of Y, version 3 of Y should be added to the collection. The dotted arrows indicate version resolution, which is explained in Section 6.4.

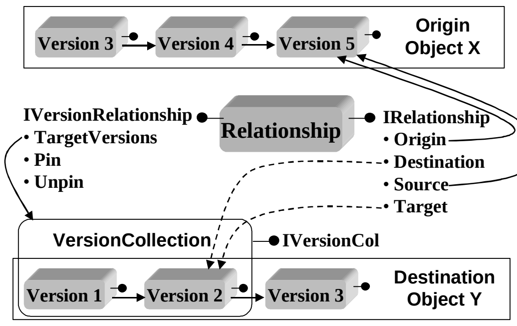


Figure 8 Versioned Relationship.

If the source version of a relationship is the origin, then one can use the method IVersionedRelationship::Pin to identify a particular target (i.e., destination) version to be the default destination for that relationship. Its usage is described in Section 6.4.

6.3 Workspace Model

To support team-oriented activities, a repository needs to let users isolate their working versions from each other and make updated versions sharable later in a controlled way. This amounts to support for long-lived design transactions. The time-tested approach to these requirements is checkout-checkin to and from private workspaces.

V2 supports workspaces. A workspace is a virtual repository that contains a subset of the objects and versions in the global repository. Each workspace is represented by an ordinary repository object, so workspaces can be grouped into collections and have custom relationships. They cannot currently be nested, a restriction we plan to relax in the future.

Versions are explicitly added to (or removed from) a workspace, thereby making them visible (or invisible) in the workspace. A version can be added to many workspaces. However, there can be at most one version of an object in each workspace. Thus, a workspace is a single-version view of a subset of the repository database.

As explained in Sections 4.2 and 5.2, a client’s operations execute in the context of a repository session, S, through which it can see the entire repository. Workspaces support the session interfaces, so a client can use a workspace W as a logical repository session. By executing operations in the context of W instead of S, the client sees only those objects that are in (i.e., were added to) W, relationships on such objects, and those relationships’ target objects that are in W. The client can use S instead of W when it wants to see the entire repository.

Note that a workspace’s support of session interfaces enhances backwards compatibility. A V1 application accesses (non-versioned) objects using a repository session as its context. By replacing its session references by workspace references, the application can still use session interfaces on those workspace objects, so no other changes are needed. The resulting application only accesses objects that are in the workspace.

While a version is checked out to a workspace, it can be updated only in that workspace. It can be checked out to at most one workspace at a time. The checkout and checkin methods amount to setting and releasing long-term locks that are stored in the repository database and are used to implement “long transactions.” A typical long transaction would add some versions to a workspace, check out the ones to modify, perform updates (under short transaction control), check them back in, and perhaps freeze them. A common usage scenario might be to populate a workspace, perhaps in a batch operation, and use it for a long period, during which many long transactions would be run.

Note that a (short) transaction is associated with a repository session, not a workspace. Therefore, a transaction can include operations on multiple workspaces, allowing complex models of sharing to be built with workspace primitives. For example, a transaction could checkin a version in one workspace devoted to testing, freeze it, and then add it to another workspace containing released objects.

6.4 Single-Version Model

Objects can be accessed without referring to specific versions, using the V1 API. There are two main reasons to continue to use this non-version interface in V2:

- backward compatibility - an application written for V1 must continue to run on V2
- simplicity – avoid burdening all tools with an understanding of the versioning model

Therefore, V2 performs automatic version resolution for non-version operations. That is, for a non-version operation, the engine accesses or returns a particular version of the object. If the operation applies to an object in the current workspace context, version resolution accesses or returns the version of that object contained in the workspace. Otherwise, if there is no workspace context, it returns the most recently created version of the object. For example, if accessed in the context of a workspace, the method get_Object(object ID) returns the version of that object that is currently in that workspace, or null if no version of that object is there. If accessed in the context of the whole repository, it returns the latest (i.e., most recently created) version of the object. Similarly, when traversing a relationship in the context of the whole repository, it gets either the pinned version in the relationship, if there is one, or the latest one, if not (e.g., the dashed arrows in Fig. 8).

Notice that an application written for V1 can run without modification on V2. Since it is written for V1, it is only able to create one version of each object and only views the objects in the context of the whole repository, so version resolution is unnecessary. Even if a V2-aware application creates new versions and moves them in and out of workspaces, the V1 application only sees the latest or pinned version of every object.

7 STORAGE MODEL

7.1 Versions and Properties

The repository engine stores properties and relationships of repository objects in a SQL database. The database has two main generic tables, the Versions table and Relationship table, which are present in every repository. There are also information-model-specific property tables that contain the properties that appear in custom interface definitions.

Each row of the Versions table contains a version's object ID, version ID, type ID of its class, frozen status, and version ID of its predecessor. Every version has one *version row* and zero or more *merge rows*. Its version row describes the version and points to its predecessor on which CreateVersion executed. A merge row describes a predecessor that resulted from executing MergeVersion. A per-row flag distinguishes version and merge rows.

The repository engine automatically generates or modifies the layout of an information model's property tables whenever an information model is created or modified. Each property table stores properties from one or more interfaces, and each interface's properties are stored in exactly one table. By default, the engine maps each interface to a separate table. However, users can override this default in the information model by mapping several interfaces to the same table. This is useful when most classes that use one of the interfaces use the other interfaces too.

Many properties of an object often have the same values for many versions of that object. Therefore, to economize on space, instead of storing a separate property table row for each version, a property table row can describe a set of versions. That is, the key of each property table is a range of versions on a branch defined by [Internal object ID, Branch ID, Start-version-number, End-version-number]. Version numbers (not to be confused with version ID's) are assigned sequentially within branch and are not exposed via the API. For a given property table row, if End-version-number is infinity, the property values apply to all versions on the branch starting from Start-version-number. Therefore, when creating a new version V on the same branch as its predecessor, no property tables need to be updated, since V's state is initially the same as its (unique) predecessor.

An update to a property P of V (denoted V.P) requires V.P's property table row to be split, since V.P now has a different value than earlier versions in its row's version range. V.P's current row is assigned the End-version-number of V's predecessor on V's branch, and a new row is inserted with V as the start-version-number. (See Fig. 9.) Also, after creating the first successor of a version, each subsequent immediate successor starts a new branch; since each property table row describes a version range on only one branch, the new branch requires a new property table row. In our usage scenarios, new branches are created infrequently, so optimizing for new versions on the same branch is appropriate.

Internal object ID	Branch ID	Start-version-number	End-version-number	P	Q
17	0	0	∞	“A”	1

Assuming version [17,0,2] is frozen and [17,0,3] is not, update P in version [17,0,3] to “B”

Internal object ID	Branch ID	Start-version-number	End-version-number	P	Q
17	0	0	2	“A”	1
17	0	3	∞	“B”	1

Figure 9 Splitting a Version Range

7.2 Relationships

Since any two versions of an origin and destination object can be related, careful design is needed to avoid a combinatorial explosion in the number of relationships between versions. V2 does this by using version ranges, similar to their use in property tables.

All relationships are stored in a single Relationship table. Its main columns are an origin version range (i.e., [Internal object ID, Branch ID, Start-version-number, End-version-number]), a destination version, and a relationship type. Each row represents part of a relationship of a given type from the versions in the origin version range to the destination version. There may be several such rows representing parts of the same relationship, because an origin version may be related to many versions of a destination. So, constructing a relationship object of a given type for a given origin version involves selecting the rows whose origin range includes the given version and creating a version collection of the destination versions (as in Fig. 8). Of course, the table can be used to traverse the relationship in either direction, so the same row represents a relationship from the destination version to the origin.

When executing CreateVersion on version V, thereby creating version V′, the repository engine ordinarily copies V’s origin relationships to V′. (This copying can be disabled by setting a flag on each relationship type definition.) As in property tables, each row of the relationship table describes a version range on one branch. Therefore, if V and V′ are on the same branch and if End-version-number is infinity for V’s origin relationships, as is usually the case, then the relationship table need not be updated to cause the relationship to be copied. However, if V′ starts a new branch, a new row is needed in the relationship table for each of V’s relationships being copied to V′. Thus, it is cheaper to copy the relationship than not. A new row is also needed when, for a given origin version, V, a new destination version DV is added. When DV is on the same branch as other destinations of V, an optimization to avoid the extra row would be to store a destination range, but this is not implemented in the current release.

Other relationship semantics that requires persistent storage is naming, sequencing, and pinning.

- The relationship table has a Name column, which contains the name of each naming relationship. An update to a name may cause a relationship range to split, just like property tables.
- In a sequenced collection, each relationship points to the previous relationship in the same collection. (Only origin collections may be sequenced.) As usual, an update to the sequence causes the relationship range to be split. When loading such a collection, the entire collection is cached and ordered by the sequencing information.
- For a pinned relationship, the ID of the pinned destination of an origin is stored in the relationship row.

For efficiency, there is one type of relationship that is not stored in the Relationship table: The relationships from a workspace to the items it contains are stored in a separate table, since this table doesn't need many of the columns of the generic relationship table.

7.3 Type Information

The type definition interfaces of Fig. 4 (`IPropertyDef`, `IInterfaceDef`, `IClassDef`, etc.) are mapped to property tables just like user-defined interfaces in information models.

In particular, property tables for type information have the same [Internal object ID, Branch ID, Start-version-number, End-version-number] key as all other property tables. However, since type information currently is not versioned, the Branch ID, Start-version-number, and End-version-number are always zero.

The type definition interfaces provide sufficient information for the engine to map accesses on properties and relationships of objects into SQL accesses on the underlying tables. For example, `IPropertyDef` has the properties `APIType`, `SQLType`, `SQLSize`, `SQLScale`, `ColumnName`, and `Flags`. And `IInterfaceDef` has the properties `InterfaceID`, `SQLTableName`, and `Flags`. Therefore, given a reference to property P of interface I, the repository engine can find the property table corresponding to I and the column of that table corresponding to P.

Since a class's type information is frequently accessed, it is cached in an optimized main memory structure that is also persisted, to avoid recomputing it every time the repository is opened. The cost is updating this structure whenever a type definition is updated, a non-trivial but infrequently-incurred expense.

8 INFORMATION MODELING

8.1 Motivation

Most database systems regard schemas as part of the application. However, repository systems include the schema, i.e., information model, as a core component, separate from the application. The main customers of a repository system are tool vendors. Tool vendors do not want to needlessly repeat the design of information models that are already well understood. Moreover, tool vendors' customers want tools purchased from different vendors to interoperate, which is possible only if they share information models. Therefore, Microsoft Repository includes a large, built-in information model, called the Open Information Model.

The Open Information Model (OIM) is a generic model that diverse tool vendors can easily extend and customize to support their particular tools. Tool functionality is dependent on the underlying information model, so an overly broad standard model can reduce some vendors' competitive advantage. A very broad model is also extremely difficult to develop, especially if it includes much detail, since it must merge conflicting requirements from a large number of vendors. Therefore, the Open Information Model (OIM) focuses on the essential functions that most tool vendors support. This lets tool vendors extend the model with features that differentiate their tools. It also circumvents the difficulty of merging conflicting requirements regarding controversial features. Over time, as the best vendor-specific concepts become more widely adopted, these concepts will migrate into OIM.

To work, this approach requires that the repository be easy to extend independently by different vendors. For this reason, information model extensibility was a primary goal of the Microsoft Repository design.

8.2 Defining and Extending Models

An information model consists of class, interface, property, relationship, collection and method definitions, as explained in Section 3.2. Generally, an information model is packaged in its own repository type library, making it easy to tell if a repository has that information model loaded and so that developers of different information models need not worry about name conflicts and the like.

A simple way to extend an information model is to define new interfaces that add new properties, relationships, and methods. One can add these interfaces to existing classes. Or, one can define new classes that support these new interfaces, as well as existing ones, if desired.

The same programming interface is used both for defining a new information model and extending an existing one. Information model definitions are implemented using ordinary data manipulation operations for repository objects. For example, to create an interface definition, I, one creates an instance of the class InterfaceDef and assigns `values` to its properties (e.g., `InterfaceID`, `SQLTableName`, and `Flags`). To create a property on I, one creates an instance P of PropertyDef, assigns values to P's properties and adds a relationship from I to P. The operations that store the new definition objects in the repository also cause the creation of new property tables and additional columns to existing property tables as defined in the model extensions. These SQL data definition operations are performed immediately before the enclosing transaction commits.

Like all repository update operations, those that create and update information models are bracketed by a transaction. Immediately after such a transaction commits, one can start creating instances of the new definitions. Thus, information model extensions can be made dynamically, without shutting down the repository engine, as is required by many older-style products.

Higher-level library functions are offered to reduce the amount of code needed for model definition. For example, there is a function to create an interface definition and fill in its properties, all in one operation. But most users don't write model definition code at all. Rather, they use the repository's graphical modeling tools based on the Unified Modeling Language to generate model definition code. These tools are described in Section 10.

In addition to defining new properties and relationships, one can extend the behavior of the repository engine by providing custom code. Useful extensions could include validating integrity constraints on properties and relationships, adding custom methods to interfaces (e.g., supporting a `Build` method on `IProject`), or storing some properties of an object outside the repository (e.g., in a file). This is done by writing a wrapper for the repository object, re-implementing the interfaces to be extended, and calling the repository engine's base implementation of those interfaces to read and write properties and relationships. (See Fig. 10.) Interfaces that are not extended are simply passed through. The mechanics of this wrapping is defined by COM aggregation, which was summarized in Section 2.1.*

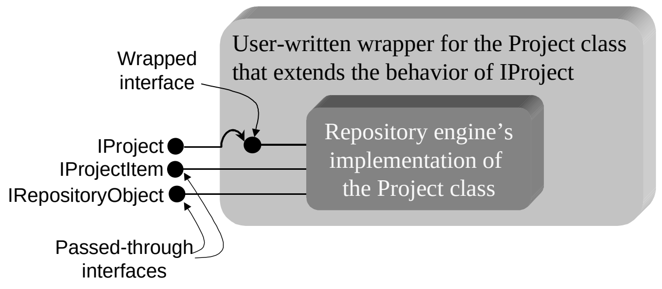

User-written wrapper for the Project class that extends the behavior of IProject
Wrapped interface
Repository engine's implementation of the Project class
IProject
IProjectItem
IRepositoryObject
Passed-through interfaces

Figure 10 Using COM Aggregation to Extend a Repository Object Class

Another form of extensibility is the ability to create new versions of interfaces — that is, new versions of information models. This is a major problem in many repository systems. Microsoft Repository supports it using the standard COM approach explained in Section 2.1:* Every published COM interface is immutable. Its interface ID identifies a contract that, once published, cannot be changed. So, to change an interface I, one defines a new interface I′. Newly written clients are built to prefer I′ but cope with I.

In this scenario, a newly written class C is built to support both I and I′, so that both old and new clients can use C. If I′ inherits from I, C can be implemented simply by defining C to the repository engine, telling it to implement I′. If I′ does not inherit from I, then C must explicitly support both interfaces. If I and I′ have overlapping semantics, such as a common property that should only be stored in one place in the repository, then one can aggregate C and, in the wrapper, map one interface's members to the other's. This mapping amounts to a procedurally implemented view.

8.3 Interface Orientation

COM is highly interface-centric. One can write programs that access objects by navigating interfaces and never know the class of which those objects are instances. The only reason to know an object's class is to create the object in the first place.

COM's interface-centric view has a profound effect on tools that share objects in the repository. To share data, tools only need to agree on interface definitions, not on class definitions.* For example, suppose we define an interface `IComponentDescription` that includes properties `Owner`, `TechnologyType` (e.g., ActiveX Control, Stored Procedure, Java applet), and `Status` (e.g., draft, unit-tested, system-tested) and includes a collection `Keywords` that is a relationship to objects that support `IKeyword`. A component reuse tool that understands `IComponentDescription` could display useful information about the component and offer keyword-based search of components. Many different tools could create reusable components that support `IComponentDescription`. For example, development tools for ActiveX controls, stored procedures, and Java applets could create objects of different classes, but all those classes could support `IComponentDescription` and therefore be visible to the component reuse tool. Thus, to support sharing between tools, interface definitions are more important than class definitions.

To share information, class definitions from different vendors support the same interfaces. However, similar classes from different vendors (such as the table definition class supported by database design tool vendors) don't need to support the same combination of interfaces and typically have different implementations of those interfaces.* This flexibility is another form of model extensibility. It has high payoff to a large vendor like Microsoft, which expects many other vendors to use its repository.

Still, vendors often need to make some assumptions about which sets of interfaces are used in combination. Otherwise, client tools would need to cope gracefully with too many different combinations of interfaces, which is expensive in development time and error prone when a large number of interfaces are in use. Therefore, an information model specifies which sets of interfaces should be implemented together. For example, it might say that if a class supports `IForm`, then it must also support `IProjectItem`. In COM, such a combination of interfaces is called a *cotype*.*

9 OPEN INFORMATION MODEL

The main goal of the Open Information Model is to provide sharing and reuse of meta-data throughout the application lifecycle, from analysis and design through implementation, deployment, and maintenance.

OIM was introduced in early 1997. Since then, new sub-models covering different subject areas have been added periodically, and the model is growing in breadth and depth.

Each sub-model is developed in collaboration with several vendors that are expert in the relevant subject area, and then reviewed for several months by a much wider audience consisting of vendors that are candidate users of the model. After processing the comments, the vendors who are collaborating on the sub-model implement prototype applications and tool integration scenarios.

In our experience, prototyping is a critically important ingredient in model development. Developing a model in the absence of usage scenarios does not work. Real usage scenarios for which the model is intended must be implemented in enough detail to flush out all of the requirements. In every sub-model we have developed, prototyping the usage scenarios has identified missing concepts and inconvenient structures that led to major modifications. We require a working prototype application, and in many cases a final application product, before freezing a model for release.

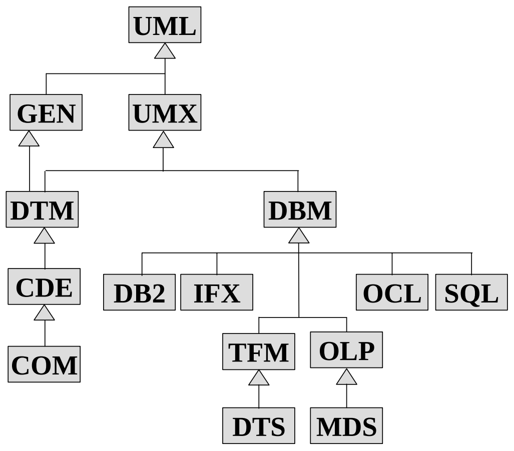
To attain the main goals of the model, sharing and reuse, it is essential that a large number of vendors agree to support it. A vendor may object to a proposed model because it fails to include a property or relationship required by an important feature of the vendor's tool. Or, if a new model is structured very differently than the persistence format currently used by a tool T, then porting T to the new model may be too expensive, and T won't use the model to share its data.

One way to make it easier to get agreement on an information model is to base a model on accepted industry standards, that is, standards that vendors are firmly committed to follow. We have followed this approach in OIM whenever such a standard is available, such as ANSI SQL, the Unified Modeling Language, and COM. Using an industry standard also saves design time, and therefore reduces time-to-market, an important business consideration.

Currently, OIM has approximately 250 interfaces and 100 relationship types. It includes a class reference model, which are the classes used by Microsoft applications. However, as explained in the previous section, the interfaces are the more important part, in that they enable sharing between tools.

Despite its large size, the model is easy to understand, because it conforms to widely-accepted standards and is partitioned into separable sub-models, each of which is in a separate repository type library and is small enough to be understood as a unit. To make it easier to find the sub-model that contains a given interface definition, each interface name includes a prefix that identifies the sub-model that defines it. For example, database model interface names begin with `IDbm`, component description model names begin with `ICde`, etc.

Currently, there are 15 sub-models, which we describe in the rest if this section. Refer to Fig. 11 for a model roadmap, which shows the primary inheritance relationships between the models. The triangular arrows are UML notation, pointing from specialization to generalization.



UML	**Unified Modeling Language**
UMX	**UML Extensions**
GEN	**Generic**
DTM	**Data Type Model**
CDE	**Component Descriptions**
COM	**Component Object Model**
DBM	**Database Model**
DB2	**IBM DB2**
IFX	**Informix**
OCL	**Oracle**
SQL	**Microsoft SQL Server**
TFM	**Transformations**
DTS	**Microsoft Data Transformation Service**
OLP	**OLAP**
MDS	**Microsoft Decision Support**

Figure 11 Open Information Model

9.1 The Unified Modeling Language

The core sub-model of OIM is the Unified Modeling Language (UML). UML is a semantic model and graphical notation for analyzing, specifying, designing, constructing and visualizing software artifacts [4], [10], [18], [19].

Initially developed by Rational Software Corporation in collaboration with many other vendors, UML is now an OMG standard. This standardization made it relatively easy to get agreement for the UML information model among the vendor community. Since most of the world's object modeling tools are evolving toward the UML standard, the UML information model provides a low-impedance path for such tools to store the fine-grained representation of models in Microsoft Repository.

In addition to its specific use for modeling tools, UML's breadth and high level of abstraction make it an excellent base model from which other OIM sub-models can inherit. It covers such concepts as type, class, component, package, diagram, method, operation, relationship, attribute, and constraint — concepts that are relevant to virtually all OIM sub-models (see Fig. 12). Having OIM sub-models inherit from UML yields the usual benefits of reuse via inheritance:

- It reduces the overall size and complexity of OIM by reusing UML concepts in many sub-models. For example, the relationship `IUmlPackage`-*Owns*-`IUmlElement` is the general containment relationship in UML. It is used to model database catalogs (`IDbmCatalog`) owning schemas (`IDbmSchema`) and tables (`IDbmTable`), COM type libraries (`ICdeTypeLibrary`) owning classes (`IComClass`) and interfaces (`IComInterface`), and data transformation packages (`ITfmTransformationPackage`) owning transformation steps (`ITfmTransformationStep`).

- It makes sharing between different types of tools simpler and more efficient. For example, a useful function in a database-oriented development tool is to generate a class definition from a table definition. Usually, this requires translating details about the table definition into details about the class. However, in OIM, since both database table (`IDbmTable`) and COM class (`IComClass`) inherit from type (`IUmlType`), many details of a table definition can be interpreted (abstracted) as details of type, which in turn are interpreted (inherited) as details of a COM class. Thus, a table definition's details can be directly interpreted as a COM class definition, without any explicit translation.

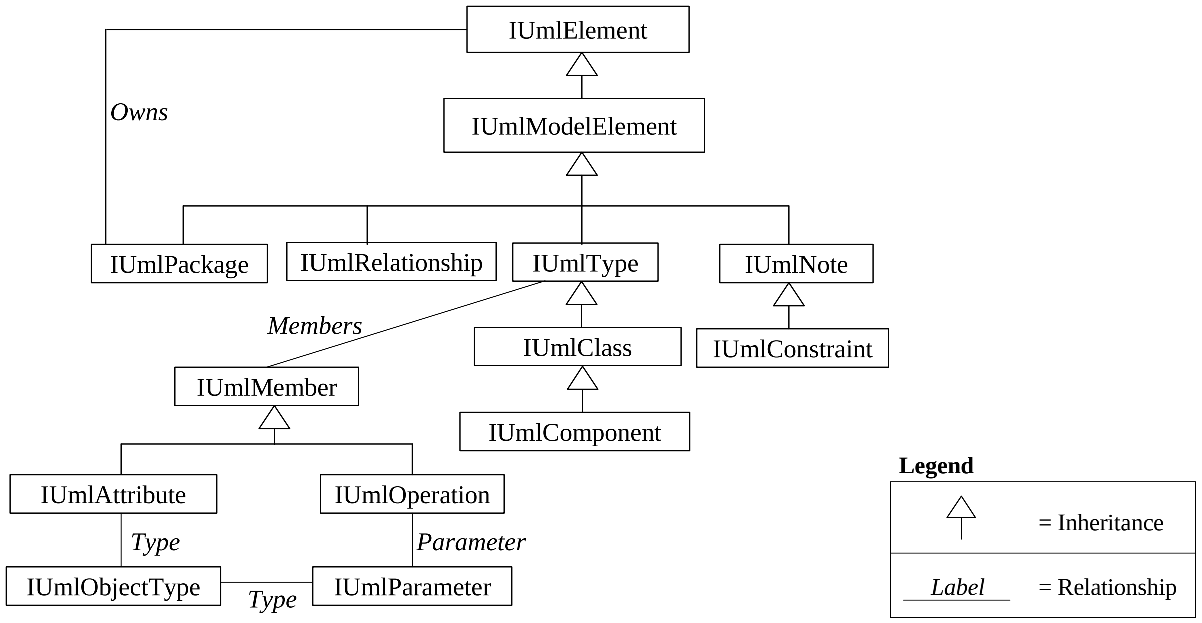


Figure 12 Key Interfaces of the UML Information Model

The UML information model in OIM was developed immediately after the original Unified Modeling Language 1.0 proposal was submitted to the OMG, and is therefore derived from that 1.0 version of UML. (Our comments about UML in the rest of this paper refer to the UML 1.0 version.) The behavioral part of UML, covering such areas as use cases, collaborations, and state machines, was immature in UML 1.0 and therefore was not included in OIM. We plan to update the UML information model to include features of the OMG's latest version of UML (currently version 1.3) soon.

9.2 Support Models

OIM includes three support models that are used in the context of most other models: the UML Extension Model (UMX), Datatype Model (DTM), and Generic Model (GEN).

OIM requires some concepts that are at the same level of abstraction as UML but are not part of the UML standard. These concepts are included in the UMX Model. For example, UMX includes a model for layout of object model diagrams (e.g., point, line, graphic element, font), information about free-standing modules (e.g., helper functions; by contrast, a UML "module" must be a method that is a member function of a type), and commonly used properties of an attribute (e.g., IsNullable, IsReadOnly, IsConstant, and MaximumLength). Rather than add these concepts to the UML information model, and thereby make it non-standard, we separated these OIM-specific UML extensions into UMX.

UML includes the concept of type, which covers the notion of abstract data type, including such things as members, nested sub-types, and templates, along with relationships to attributes and parameters of that type. However, it does not include the details of commonly used primitive types. Therefore, we added the DTM model to OIM, to refine UML type with numerous common data types, including atomic types (string, integer, float, binary, decimal, pointer, etc.) and constructors (structure, union, array, and enumeration). These types are further refined in other models, such as the COM and database models.

The generic information model (GEN) is a catch-all for concepts that are equally applicable to all models and have no natural home elsewhere in OIM. Examples include help information, summary description, keyword, email address, menu, and icon. Many of these concepts in GEN could be the basis of entire subject areas, such as Catalog, Person, and

Graphical User Interface. However, until we develop these subject areas in some detail, it is more convenient to group such concepts in GEN rather than proliferate tiny sub-models.

9.3 Database Information Model

One of the main applications of Microsoft Repository is database schema management for database design, schema reuse, and data warehousing. To support these applications, OIM includes many database-oriented sub-models. The core sub-model is called DBM (database model).

DBM specializes UML to include the concepts found in standard SQL data definitions and similar types of formatted data models. Thus, it includes the concepts of database catalog, database schema (e.g., all objects owned by a user), column set (an abstraction of table and view), table, view, constraint, trigger, and column. Most of these concepts inherit from UML: catalog and schema inherit from UML package (UML's general container concept); column set, table, and view inherit from UML type; constraint inherits from UML constraint; trigger inherits from UML method; and column inherits from UML attribute (a structural feature of a UML type). Some DBM concepts do not have natural analogs in UML, such as key and index, so they inherit from the very abstract UML concept of model element.

For the most part, DBM focuses on logical database concepts. However, it also includes some physical database concepts since they are needed in nearly all usage scenarios. For example, DBM includes the concepts of index (since many reusable database designs include index definitions), data source (a provider of database services to which a client can connect), and database connection (which, together with data source, provide a bridge between the logical and physical worlds). In DBM's class reference model, many DBM interfaces have both a logical class that represents the data definition in the abstract and a deployed class that represents an instantiation of the definition at a node. The latter is obtained by having the class support IUmlComponent, which represents a deployed object and has a relationship to the node on which it is deployed. For example, the DbmTable class implements IDbmTable and the DbmDeployedTable class implements IDbmTable and IUmlComponent.

DBM includes sub-models that capture data definition capabilities that are unique to popular database systems, such as IBM's DB2, Informix, Microsoft SQL Server, and Oracle. Although most SQL database systems incorporate many proprietary extensions to their data manipulation languages, they differ very little in their data definition capabilities. Therefore, these sub-models are quite small, typically adding only a few interfaces, each with just a few properties. For example, the Oracle sub-model includes the concepts of cluster (for multiple tables with common key mapped to the same storage area) and sequence (a special data type for sequence numbers). The DB2 sub-model includes special options on views and functions. And the Microsoft SQL Server sub-model includes special options on index definitions.

9.4 Data Warehousing Models

The DBM model is extended in two dimensions to support data warehouse applications. First, it adds special types to support multi-dimensional (i.e., OLAP) data. Second, it adds details of data transformations used to populate data warehouses. Each of the subject areas is split into two sub-models — one for generic concepts that are relevant to all products that offer the given capability and one for specialized concepts that are specific to Microsoft products. Specializations of the generic models could be done for other third party products, as we did for the DB-product-specific specializations the generic DBM model.

9.4.1 OLAP Models

The OLAP model (OLP) captures descriptions of data cubes. OLP's basic concepts are all specialization of DBM: OLP catalog inherits from DBM catalog; OLP store inherits from DBM column set; and OLP measure (or fact) inherits from DBM column. OLP cube, aggregation (partial roll up), and partition in turn inherit from OLP store. OLP has some unique concepts that do not have natural counterparts in DBM or UML, such as dimension (of a star schema or cube), dimension hierarchy, and mapping between dimension hierarchy and store. Therefore, these concepts inherit from the very abstract UML concepts of package and model element.

One OLP specialization is the Microsoft Decision Support Services (MDS) model, which describes detailed attributes used by the OLAP server that ships with Microsoft SQL Server 7.0. This model simply adds properties to generic concepts in the OLP model.

9.4.2 Transformation Models

When moving data from production databases into a data warehouse or data mart, data typically needs to be scrubbed to filter out erroneous data and transformed into a more suitable form for querying. Source and target schemas and transformations between them is valuable meta-data to store in a repository. It can be used to support transformation design, impact analysis (which transformations are affected by a given schema change), and data lineage (which data sources and transformations were used to produce given warehouse data).

The transformation model (TFM) captures information about compound data transformation scripts. Individual transformations (which inherit from UML method) have relationships to the sources and targets of the transformation. Some transformation semantics can be captured by constraints and by code-decode sets for table-driven mappings. Transformations can be grouped into a task (the unit of atomic execution, which inherits from UMX module). Tasks are invoked by steps (the unit of scheduling) within a top-level script called a package (which inherits from UML package). (See Fig. 13.) Package executions are also modeled. Thus, each data warehouse data item can be tagged by the identifier of the package execution that produced it, to enable traceability of the data item's detailed data lineage.

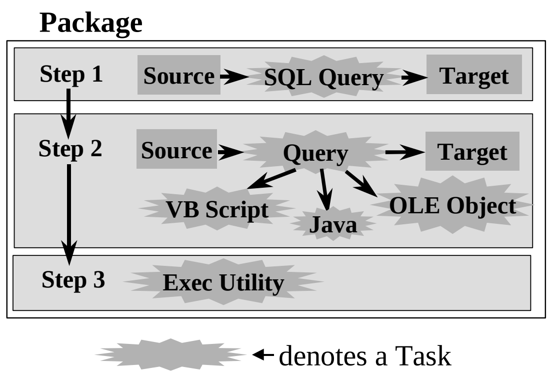


Figure 13 Grouping Transformations

TFM is specialized to the DTS model to support the SQL Server Data Transformation Service. Primarily, the DTS model captures detailed properties of particular kinds of TFM script objects. For example, it includes properties of data pump tasks, transfer object tasks, and send mail tasks, all of which inherit from TFM transformation task, and of transformations, steps, and step executions.

9.5 Component Description Model

The Component Description Model (CDE) specializes the UML concepts of type (specification) and component (executable) to support tools for component based development and reuse, such as the Visual Component Manager in Visual Studio 6 (see Fig. 14).

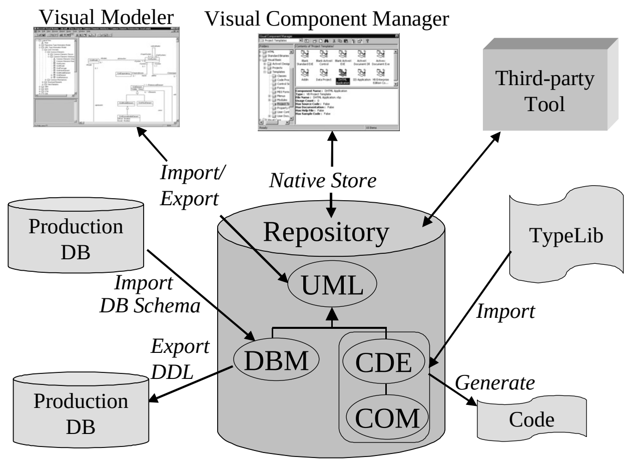


Figure 14 Exposing Shared Components Using CDE

CDE defines component as “a software package that offers services through interfaces.” This is meant to capture the perspectives of a component as the unit of packaging and delivery, provider of services, and encapsulation boundary.

CDE extends UML with details that are common to all popular object models, such as COM, CORBA, and Java Beans. At the specification level, it includes such concepts as pre- and post-conditions, event sources, operation signatures, idempotence, default interfaces, and required vs. optional functionality. At the executable level, it includes descriptions of resources, licensing, and installation aspects of a component.

The COM model specializes CDE with detailed properties that are found in COM type libraries, such as class and interface ID’s, threading model, aggregatability of a class, whether an interface is dual, etc. Utilities exist to import a COM type library as instances of the COM model, and export a COM model as an interface definition that can be compiled into a COM type library.

10. MODEL DEVELOPMENT KIT

Most customers of Microsoft Repository are tool vendors who need to extend OIM to customize it for their particular product. To simplify this task, as well as our own task of defining and implementing OIM, Microsoft Repository includes a tool kit for using and extending OIM and developing information models of one’s own. It includes tools for defining models and for generating model installation programs, documentation, and tests.

The tool kit includes all OIM sub-models in UML format, so a model designer can conveniently view the details of OIM and use cut-and-paste to reuse OIM interfaces when producing a custom extension.

The first step is to define an information model in UML and save it in Microsoft Repository (see Fig. 15). One can then use the model checker, model generator, and documentation generator that all run on the stored representation of the model. The model checking tool identifies structural errors in the model: missing properties, partial definitions, inconsistent definitions, duplicate or invalid names, etc. It optionally generates globally unique ID’s for new interface and class definitions. It is important that this be done during model definition so that the finished model will use the same ID’s in every repository where it is installed.

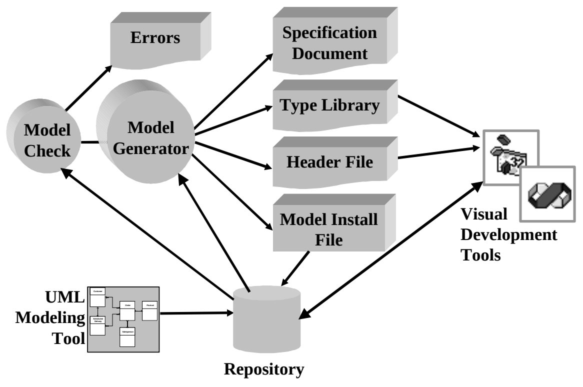


Figure 15 Model Development Tool Kit

After an error-free execution of the model checker, the model generator can be used to produce a model installation script. The script is a data file that contains a definition of the model and is used as input to a generic model installation program that is part of the tool kit.

The tool kit also includes a documentation generator that produces a document in Rich Text Format (RTF), containing definitions of all interfaces, relationships, and classes defined in the information model.

11. SUMMARY

In this paper, we described the design and implementation of Microsoft Repository, focusing on three main capabilities:

- its object model, integrated with COM, to support declarative definition of extensible information models
- its version and workspace model, to support design objects that evolve over time;
- and its Open Information Model, to enable the sharing and reuse of meta-data throughout the application lifecycle.

Over a million copies of the first version shipped with Visual Basic 5.0. The second release is now available in Visual Studio 6.0 and SQL Server 7.0, where it is applied to component management and data warehousing. Over 75 third parties are shipping tools that use the repository for these and other applications. We expect future versions of the product to expand the set of capabilities and, thereby, the types of applications which it can satisfy.

Acknowledgments — The design of the Microsoft Repository API was a joint effort between Microsoft Corp. and Sterling Software, originally conceived by David Vaskevitch and Keith Short. We're grateful for their sponsorship. We learned much from early contributors to the design effort, including John Cheesman, Bill Dawson, David Maier, Betty O'Neil, Melissa Waldie and Laura Yedwab . We especially thank Brian Harry and Jason Zander for their major contributions to the V1 engine design.

V2 was implemented in partnership with PLATINUM Technology, Inc., who contributed to both the engine and information model. We thank Murat Ersan, Jayaram Mulupuru, Ragini Narasimhan, and Daniel Reib, for their help with many aspects of the design and implementation, as well as Sam Abolrous, Matthew Burke, Jim Hance, Gary Miao, Tad Price, and Brian Trede for other contributions to the V2 effort. Finally, we thank Patrick Cross, Joe Maguire, Shawn Oberst, Brian Welcker, and the many companies and individuals who have contributed to the Open Information Model, regrettably too numerous to list here.

REFERENCES

[1] Berenson, H., P. A. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proc. ACM SIGMOD 1995*, ACM, N.Y., pp. 1-10 (1995).

[2] Bernstein, P.A. Repositories and Object-Oriented Databases. In *Proceedings of BTW '97*, Springer, pp. 34-46 (1997). Reprinted in *ACM SIGMOD Record*, **27**(1) (March 1998).

[3] Bernstein, P.A., U. Dayal. An Overview of Repository Technology. In *Proc. of 20th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, San Francisco, pp. 705-713, (1994).

[4] Booch, G., J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, Reading MA (1998).

[5] Cattell, R.G.G., D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, D. Wade. The Object Database Standard: ODMG 2.0, Morgan Kaufmann Publishers, San Francisco, CA, (1997).

[6] CDIF Technical committee. CDIF standards and related documents are available at http://www.cdif.org

[7] Chou, H.-T. and W. Kim, "A Unifying Framework for Version Control in a CAD Environment." In *Proc 12th International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, San Francisco, pp. 336-344 (1986).

[8] Constantopoulos, P., M. Jarke, J. Mylopoulos, Y. Vassiliou. The Software Information Base: A Server for Reuse. In *VLDB Journal*, **4**:1-43, Boxwood Press, Pacific Grove, CA, (1995).

[9] Dahanayake, A. *CAME: An environment to support flexible information modeling*. CIP-Gegevens Koninklijk Bibliotheek, Den Haag, The Netherlands, ISBN: 90-9010742-8 NUGI 851 (1997). Available from the author: A.N.W.Dahanayake@duticai.twi.tudelft.nl.

[10] Fowler, M., and K. Scott *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, MA (1997).

[11] Geiger, K. *Inside ODBC*. Microsoft Press, Redmond, WA (1995)

[12] Gray, J., R. Lorie, G. Putzolu and, I. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *Readings in Database Sys*, 2nd Edition, Chapter 3, Michael Stonebraker, Ed., Morgan Kaufmann Publishers, originally published 1977, (1994).

[13] Jarke, M, R.Gallersdörfer, M.A. Jeusfeld, M. Staudt, S. Eherer. ConceptBase - A Deductive Object Base for Meta Data. In *Journal on Intelligent Information Systems*, **2**(4):167 – 192 (March 1995).

[14] Katz, R.H. "Toward a Unified Framework for Version Modeling in Engineering Databases," *ACM Computing urveys* 22 (4): 375-408, (Dec. '90).

[15] Microsoft Corporation. COM web site, http://www.microsoft.com/com

[16] Microsoft Corporation. Microsoft Repository web site, http://www.microsoft.com/repository

[17] Microsoft Corporation. Microsoft Foundation Classes and Templates. In Microsoft Visual C++ Documentation, Microsoft Developer Network for Visual Studio 6.0, 1998.

[18] Object Management Group. *OMG Technical Library.* At http://www.omg.org.

[19] Rational Software Corporation. Unified Modeling Language Resource Center. At http://www.rational.com/uml

[20] Rogerson, D. *Inside COM*. Microsoft Press, Redmond, WA (1997).

[21] Rumbaugh, J. "Controlling Propagation of Operation using Attributes on Relations," In *Proc. of the Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*, pp. 285-296 (1988).

[22] Sciore, E., "Versioning and Configuration Management in an Object-Oriented Data Model," IN *VLDB Journal 3:* 77-106 (1994).

[23] Vaskevitch, D., *Client Server Strategies: A Survival Guide for Corporate Reengineers.* IDG Books, 1995.

[24] Wakeman, L. and J. Jowett. *PCTE - The Standard for Open Repositories*. Prentice-Hall, (1993).

[25] World Wide Web Consortium. *XML Language Specification* and related documents. At http://www.w3c.org/TR/REC-xml.

[26] Zdonik, S.B. and D. Maier. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann Publishers, San Francisco, (1990)