Data Models in Object Management Systems

Abstract

During recent years, several research efforts in the area of software development environments have focused on the provision of uniform Object Management Systems (OMS) as a framework for tool integration and communication. This paper summarizes discussions of an OMS Workshop on the issues that arise in defining an appropriate data model for an OMS.

1. Introduction

An OMS is responsible for administering objects and the plethora of information about these objects, their properties and interrelations, as they are created, modified, and possibly deleted during software development. The term "object" is used here in a generic sense; while data ultimately is decomposable into individual bits and bytes, it needs to be aggregated into more comprehensive units in order to be manageable and to be operated upon at a suitable level of abstraction. We refer to these units as objects, without necessarily implying any connotations arising from object-oriented design methods. A precise definition of what constitutes an object is largely dependent on the design of a particular OMS and the type model applied by the user to objects in this system.

2. The Nature of Objects

In defining the nature of objects in an OMS, we face numerous issues:

- What information is agglomerated in objects ? Do different contexts refer to different subsets of this information ?
- Is there a need to compose objects from smaller objects ?
- What access controls are required on the objects and the individual pieces of information in or about an object ?
- What operations are meaningful on objects and the information in or about them ? Should the set of applicable operations be tailorable to different applications ?
- Does the nature of objects change over their lifetime ?
- Is the information in or about objects to be grouped physically, perhaps for efficiency or OMS distribution reasons ?

While the above list is far from complete, it already shows the many facets to be considered in defining the nature of objects for an OMS. The problems of *object granularity* , and of *composite objects* are a recurring theme in addressing these questions.

2.1. The Problem of Granularity

The objects administered by the OMS may have internal structure whose details are unknown to the OMS; this is primarily a consequence of a trade-off between providing generic OMS support down to the level of primitive data types and utilizing possibly more efficient special-purpose operations on the information content of more complex objects. The point of transition from the OMS data model to the data model applied to the individual objects delineates the choice of OMS granularity. For example, the granularity could be chosen so that the administered objects are host files; the OMS would then act as an administrator of files (tracking interrelations and properties of files in the OMS data model with considerably more expressive power than traditional file management systems in operating systems), while input/output packages operate on the contents of the objects and apply their respective data models to these contents.

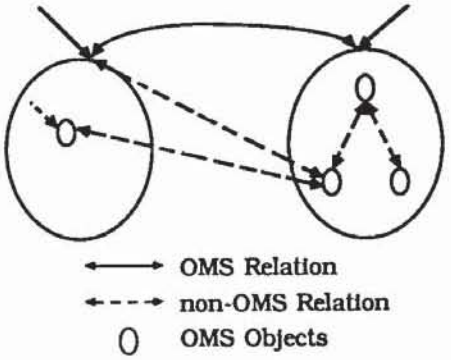

Figure 1: OMS and non-OMS relations

If the OMS granularity is chosen to be at a coarse level, problems arise in practice from the need to relate sub-granular data items within different OMS objects. In practical terms, this need translates to references from data within an object to another object or to data within another object. Figure 1 displays this situation, which is typical, for example, in supporting libraries of compilation results as done for Ada. The main problem caused by such sub-granular references is maintaining the consistency of the information; as objects get modified, existing sub-granular references to them may no longer be valid logically or representationally. However, since the OMS is unaware of such references, the necessary consistency checks must be relegated to tools that understand both the OMS data model and the data model of the objects. Consistency enforcement by the OMS in this regard is not possible.

2.2. The Problem of Composite Objects

For the administration of objects, it is often desirable to aggregate existing objects into a *composite object* to be treated as a single entity in some circumstances while, in others, the component objects are treated as separately identifiable and accessible entities.

With composite objects, a problem comparable to the one of sub-granular references exists: The OMS is now aware of relationships among components of different composite objects, but it is far from obvious how the consistency rules can be conveyed to the OMS, which ensure that relations in which components of composite objects partake are updated consistently when composite objects are created, copied, or

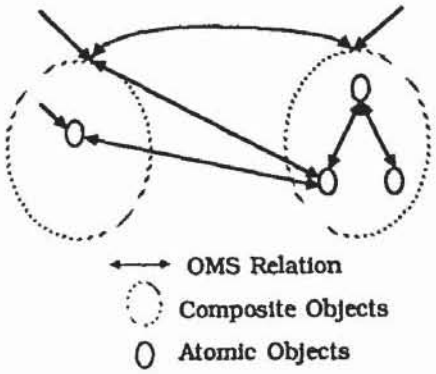

Figure 2: Composite Objects

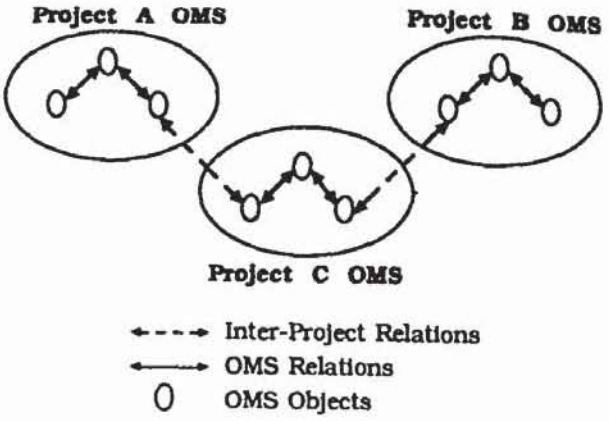

Figure 3: Project Specific OMS

modified as a whole. A picture of the situation is given in Figure 2, which is remarkably similar to Figure 1. It takes little imagination to conclude that the need for composite objects increases with decreasing coarseness of the OMS granularity. It also becomes clear that, by refining the granularity of OMS objects down to a level at which no more sub-granular references across object boundaries exist, the problem of sub-granular references has been eliminated by mapping it to the problem of relations among components of composite objects. Largely unsolved important issues in replacing coarse object granularity by composite objects are the ramifications on access control and synchronization, since applying these mechanisms at the level of individual fine-grained objects carries a significant space and performance cost. Respective "whole-sale" operations on composite objects with inheritance semantics for their components are needed to achieve acceptable performance and user convenience. Other unsolved problems are the consequences of the support for composite objects on typing and type evolution mechanisms in an OMS.

3. The Choice of OMS Scope

In designing an OMS, the scope of its applicability needs to be considered. On the one hand, one can design a single OMS to support the management of all objects on a system and imprint project management structures and policies on the object base in terms of OMS access control and typing facilities. On the other hand, one can design one or more OMS to create multiple, distinct object bases, so that only the objects relevant to a given project are administered within a project-specific OMS. The former approach poses a number of stringent requirements on discretionary and mandatory access control and on object typing approaches in the OMS to accommodate the coexistence of multiple projects in a single OMS base. The latter approach arguably may imply lesser requirements in these areas, but creates barriers for reuse of objects across projects. In order to support such reuse, either *import-by-copy* or *import-by-reference* mechanisms across project-specific OMS bases are necessary. Both these mechanisms face considerable challenges in coping with change propagation, when exported, imported or referenced data are modified. In the case of cross-project references, the picture that emerges in Figure 3 is quite similar to the figures shown earlier for the problems of object granularity and composite objects.

From these findings, one is led to conclude that the ideal OMS exhibits the following characteristics with respect to the administered objects:

- Granularity of objects can be reduced to a level at which no sub-granular references are necessary.
- A sufficiently rich set of object composition paradigms is available.
- The OMS is universally applicable to objects across projects, but tailorable to reflect project boundaries.

Unfortunately, OMS technology to-date has not been able to fulfill these (and other) criteria in an implementation that also satisfied performance requirements for use in real software production.

4. Typing of Objects

Typing of objects in programming languages is a well researched area and generally considered to be of significant benefit to software engineering. Comparatively little work has been done to develop accepted typing models for the objects that persist beyond the execution of a single program. Traditionally, objects lost all type protection and type information, once they crossed the boundary between the creating or accessing program and the operating system. Quite obviously, it would be desirable to extend the protection of typing and its influence towards good software engineering to those *persistent* objects as well. By defining types and treating persistent objects as instances of these types, numerous benefits accrue for the accessibility of the stored information and for the protection against accidental and malicious application of inappropriate operations to objects.

Typing in an OMS applies not only to the information in an object, but also extends to the information about the object, i.e., its properties described in attributes and its interrelations with other objects. We use the term *object type* in this broadened sense. Alternatively, one could speak of the *object base schema* as the sum of all object types to reflect the high degree of interconnectivity among the objects and related constraints that arise in an OMS.

A number of objectives need to be satisfied by a typing model in an OMS:

- expressing the association of properties with the instances of a type;
- expressing the association of (some) operations with the instances of a type;
- expressing constraints on instances of a type;
- enforcement of constraints in maintaining OMS consistency with the type model;
- creation of views to tailor visibility of properties by users and to resolve naming conflicts.

Depending on the choice of a particular typing model, some of these objectives may map into each other: for example, the association of properties or operations with instances of a type may well be viewed as the enforcement of a constraint limiting applicability of operations to objects. Similarly, relations among instances of object types either can be viewed as properties of instances or their utilization can be regarded as operations applied to the involved instances.

The cited objectives address the need for declarative information that assists the user in determining the precise nature of objects and the availability of information in and about these objects and of meaningful operations on them. They also address the need for conveying semantic information to the OMS that allows the enforcement of constraining rules for operations on objects. Further, they address the need for voluntary or enforced information hiding, so that users are not overwhelmed by a flood of information available but irrelevant to their momentary needs. Finally, to support a paradigm of composition of independently developed tools operating on the same objects in the OMS, one must allow that these tools have different views of the available properties and operations of a given object type. A mechanism is needed to reconcile naming differences and to resolve naming conflicts in the combined views of these tools.

Typing models in programming languages traditionally consist of:

- primitive "built-in" types (e.g., integer, boolean)
- operations to create types (e.g., arrays, records, classes)
- operations on instances of types (e.g., creation, deletion, access)
- relations between types (e.g., type derivation, type specialization) and implied semantics

There seems to be no reason why similar models could not equally be applied to typing persistent objects, although some extensions are likely to be desirable. Notably absent from these traditional models is the capability to specify more encompassing constraints on interrelations of instances of these types. The responsibility for such enforcement has been typically left to the user of the programming language (except for some efforts of integrating specification or assertion sublanguages into programming languages). Also absent in many typing models for programming languages is a mechanism for views and name conflict resolution, presumably on the assumption that, for a single program, a-priori coordination of these views into a single type definition or definition hierarchy is a reasonable expectation. Mechanisms for view creation and name conflict resolution have been developed primarily in the data base area.

Another desirable extension to the conventional model is the capability to add operations and other properties to a type definition in an incremental fashion (without affecting existing tools), rather than being forced into a closed a-priori definition of all such properties. It remains an open question whether such an incremental approach should apply throughout the entire life-time of the type definition or whether a closing of the definition is appropriate at some stage.

4.1. The Problem of Enforcement of Object Typing

In programming languages, object typing rules are primarily enforced at compile time by diagnosing the application of illegal applications of operations to objects of a given type. At run time, checks can be performed to prevent violations of constraints on the value of objects. To ensure early detection of errors, violations of typing rules for persistent objects should ideally also be caught at the time of compilation of programs that access persistent objects.

Since the types of persistent objects are generally not known a-priori to the compilation of the accessing programs, the programs can only express a *type ezpectation* for existing persistent objects. Some run-time validation of the type expectation against the actual type of an accessed object is necessary. An appropriate mapping of the OMS typing model to the typing facilities of the language in a binding of the OMS interfaces, or a direct integration of the OMS typing model into the language, can utilize the expressed type expectation to limit the operations available on the object (e.g., by equating the type expectation with an abstract data type of the language). It thereby may be possible to reduce the need for run-time validation to be performed repeatedly for each operation on the object. Generally, the latter is necessary if the OMS interfaces are accessible without utilizing a specific language binding or if one wants to safeguard the integrity of the OMS base against malicious breaches of the typing rules of a given programming language. Despite the need for such repeated run-time checks, a mapping of the OMS typing model into the typing model of a programming language is desirable to detect some error situations at compile time (even if eventual run-time checks cannot be avoided). A direct integration of the OMS typing model into a programming language, on the other hand, causes obvious problems in multi-language environments.

Lastly, typing rules and constraints could be enforced explicitly by checks in the executable code of tools. Since the danger of accidental omission of such explicit checks by the user is high, the majority of such checks should be relegated to the OMS and performed implicitly.

4.2. The Problem of Object Type Evolution

In programming languages, the general assumption is that any changes in type definitions are sufficient grounds for at least partial recompilation and relinkage of programs using such types, so that all objects of the type comply with the modified definition. In software engineering environments, this assumption is unreasonable in this generality, since a large number of objects of the type and tools operating on those objects may already exist. It is not feasible to mandate a recompilation of all such tools and an explicit migration of all objects to conform to the modified type prior to resuming normal operations. Different mechanisms are needed that allow certain modifications to be made to type definitions and yet allow continued existence of objects of the previous version of the type definition and of tools operating on old and new objects of the type.

The mechanisms for type evolution are tightly linked with the mechanisms for migrating the object base to the evolved type definitions and with the mechanisms for type binding of objects. Different type evolution models may cause differences in the model of migrating instances of types to more evolved definitions and in formulating compatibility rules between type expectations and actual object types. Three prevalent models of type evolution in OMS designs are:

(1) the specialization/generalization model: it creates new types by derivation from existing types. If the new type has lesser capabilities or weaker constraints than the old types, we speak of generalization. If the new type has

more capabilities or stronger constraints, we speak of *specialization*. Type binding is such that instances of specializations are always consistently readable under a more general type expectation. To the extent that the differences between two types are the presence or absence of properties unrelated to other properties (rather than a difference in the strength of constraints on properties), an instance of the more specialized type can be read and written under a more general type expectation. In practice, the specialization/ generalization models allow for a graceful additive evolution of the type definitions, for the introduction of generalized types for hitherto unrelated more specialized types, and for a gradual non-mandatory migration of objects from more general type definitions to more specialized ones and vice versa without unduly affecting the operability of existing tools.

(2) the type versioning model: in this model, each object has exactly one type definition under which it can be handled by tools. Such type definitions can be versioned; migration rules are provided that control the evolution of objects from one type to another version of the same type. Here, tools are affected by changes in the type definitions, unless compatibility rules between type expectations and actual types similar to those of the specialization/generalization models are defined.

(3) the in-place modification model: Here, objects have a single type definition. Changes to the type definition are associated with implicit realignment semantics for the objects and quite possibly a requirement for a "lazy" recompilation of the object base and of the tools operating on it.

In the cited three models, each instance uniquely identifies its type as given by its creation or subsequent migration to a related type. They differ mainly in the migration rules for objects and in the compatibility rules that allow existing tools to continue to operate on all objects that still satisfy its expectations of properties and constraints, regardless of the specific type binding of the object.

It can be surmised that this model of instances uniquely identifying a type, combined with compatibility rules to allow alternative type expectations, may be an unfortunate paradigm. Instead, it might be preferable for the purposes of type evolution, if the type of an object were determined by *type predicates* over the properties of the object. Each object may satisfy many type predicates. Changes to the properties of an object may implicitly cause it to assume a different set of types by now satisfying their type predicates instead. Type expectations would be satisfied if the properties of the object satisfied the respective type predicate. This alternative model eliminates the OMS problem of type evolution and corresponding object migration but, in order to prevent a chaotic evolution of objects, needs highly expressive formalisms to impose constraints on the circumstances in which changes to object properties are allowed.

4.3. Coexistence of Multiple Type Models

In examining the constituents of type models, a hierarchy can be defined, in which different alternatives can be chosen at each level. The difficulty of integrating information expressed in type models that differ in their choices at some level decreases substantially with each such level.

These levels are:

- level 1: data model, family of type models, e.g., ERA
- level 2: specialized data model, built-in OMS semantics, e.g., CAIS ERA
- level 3: data description language (DDL)
- level 4: schema definition(s) in DDL
- level 5: instances of types

Level 1 defines the overall data model, e.g., the entity-relationship-attribute (ERA) model, that serves as the common framework over which subsequent levels are built. It defines a meta-schema that delineates the domain of discourse without imposing any additional semantic constraints. If two typing models differ at level 1, then the difficulty of integrating information expressed in such different models is extremely high.

Level 2 augments the overall data model with more specific restrictions. Possibly some rudimentary semantics built into the OMS are expressed at this level. An example of a level 2 augmentation to an ERA model is the CAIS ERA model, in which some restrictions on the general ERA model are expressed and built-in semantics are provided that allow a representation of type definitions in terms of the basic ERA model. The latter communicates user-defined semantics of type definitions to the OMS in a self-descriptive fashion. If two typing models differ at level 2, then the difficulty of integrating information expressed in such different models is quite high. Some predicates may be decidable based on the common meta-schema of level 1.

Level 3 defines the language in which the user-supplied type definitions are expressed. If two typing models differ at level 3, then the difficulty of integrating information expressed in such different models depends significantly on the functionality of a common level 2. If level 2 provides a self-descriptive method of representing type information and the DDL is merely an external means for communicating this information to the OMS, then integration of information is relatively straight-forward for the OMS. Users may have some problems to relate the results of such integration back into an integrated DDL representation, in particular if the differences of the DDLs are not merely a matter of syntactic sugaring, but impact the expressiveness of the respective DDL. If level 2 does not provide a self-descriptive capability and the DDL is the primary means for integration, then the difficulties are probably as high as on level 2, when information expressed in two type models with different DDL is to be integrated.

Level 4 utilizes the DDL to provide the schema definitions that describe the specific types of objects, their properties, operations, and interrelations. It is to be expected that tools developed independently may rely on different, but overlapping type descriptions for the same objects. Suitable OMS mechanisms (e.g., views) must exist to integrate such types and reconcile any conflicts. Thus, differences at level 4 are a quite necessary part of the OMS support, rather than an avoidable complication.

Level 5 deals with the representation of objects as instances of types defined at level 4. Here, differences are to be expected, in particular, if the OMS base is distributed across heterogeneous host systems. With suitable abstraction mechanisms for accessing the objects, representational differences must be hidden from the tools utilizing the OMS.

We conclude that, for information integration purposes in an OMS, uniformity of the first three levels would be highly

desirable, while the coexistence of different approaches at levels 4 and 5 needs to be accommodated.

4.4. Universal Vs. Specialized Type Models

While the preceding section examined multiple type models from the viewpoint of information integration, it ignored the question of appropriateness and efficiency. The issue therefore remains whether substantially different type models need to be supported by an OMS on those latter grounds.

In particular, is one sufficiently general typing model functionally adequate to address the needs of the users? Or is it necessary to permit multiple typing models to be applied? In the latter case, the transition from one typing model to the other could occur at several different places:

- It could occur at the granularity boundaries of the OMS, e.g., while files are administered under the typing model of the OMS, their contents could be dealt with under the typing models provided by programming languages and their input/output capabilities, in particular by existing packages that implement a specific type model (e.g., SQL, GKS, IDL, or IRDS bindings). In this case, the responsibility of the OMS ends after ensuring that the correct typing model is chosen for handling the contents of the objects. The problem of sub-granular references remains, but all other cited problems of integrating multiple type models do not arise, since their object domains are disjoint.
- Alternatively, in the case of project-specific OMS bases, different type models could be applied to different such data bases. This would, of course, substantially aggravate the already discussed problem of inter-project references.
- Finally, in a single OMS base, the set of administered objects could be "overlayed" with multiple type models, one of which is selected for each application, based on the appropriateness of the respective typing model.

The problems with multiple type models are two-fold: first, consistency constraints that involve predicates expressed in multiple type models are exceedingly hard to formulate, in particular, if equality predicates over object references are involved. Second, the implementation effort for OMS support of multiple type models can be orders of magnitude more difficult than support for a single type model.

The problems with a single type model are mainly those of power of expressiveness. Properties easily expressed in a specialized typing model may well be difficult to state or only inefficiently implementable in such a universal and generic type model. Nevertheless, we conclude that a single, sufficiently general and adaptable type model is presently the most promising and desirable approach in addressing the various problems in OMS design.

5. Conclusions

In examining the research and industrial practice in the area of data models for OMS, we find that significant progress has been made in recent years in understanding the problems and providing some proto-typical solutions, but that a large set of issues remain to be addressed truly satisfactorily, e.g.,

- definition of requirements for the OMS typing model
- selection of appropriate typing models
- evolution of type definitions, data bases, and toolsets
- dynamicness of schemas and the impact of incremental changes
- deletion semantics in type definitions
- access control in a global OMS
- access control and synchronization at fine OMS granularity levels
- consistency checking; triggering and notification mechanisms
- boundary between the OMS and the programming language
- non-traditional execution models for typing enforcement at compile-time

This large and certainly incomplete set of unresolved problems shows the immaturity of the field and the need for significant research in the OMS area.