Abstract

A precondition for integrating object-oriented and relational technologies is an integration of their respective models. Once this is achieved, one naturally asks whether technology developed within the special contexts of the original models can be used in the context of the unified model. Due to space restrictions, this position paper outlines a general approach to unifying the models, and leaves for the panel discussion a consideration of problems seen for integrating technologies.

Unifying OO and Relational Models

Danforth and others have formalized a model that unifies OO and relational approaches [Dan92]. The model was developed to support static typechecking for the rule-based RDL/I database system [Kier90]. In the context of this discussion, the essential features of the resulting model are:

- ADTs (abstract data types) provide the sole mechanism for encapsulating data and operations.
- An inheritance ordering on ADTs reflects common abstract interfaces (not necessarily common implementations) and supports static typechecking.
- There is a distinction between “value” and “object” ADT instances. A value is always owned by a single containing data structure, whereas objects have lifetimes and identities independent of any containing structure, and can be shared by multiple structures.
- Primitive data types provided by the model include atomic data (such as integers and character strings), and data structures (such as tuples and sets). Instances of primitive data types are always values. An ADT is implemented using a primitive data type to store its state (e.g., a tuple is often appropriate for this purpose).

Relations in the Model

A relation (as in the traditional relational model) is modeled as an instance of a “relation ADT,” whose state includes a set of tuples and other data (likely including ADT instances) used to implement consistency constraints, locking information, indexes, cursors, etc. The operations provided by relation ADTs support adding and deleting tuples, opening cursors, retrieving tuples, etc.

Operations that create new relations (e.g., the traditional select, join, etc.) are provided as primitives by programming languages based on the model. For example, an Object Oriented SQL might be provided (also, see [Dan92] for an example based on the RDL/I language [Kier90]). Programs written in or interactively interpreted by these languages use and create relations through the abstract interfaces provided by relation ADTs.

A relation instance in the model may be a value or an object. Top-level relations that have identity and persist between program executions will of course be objects. Relations returned as the result of queries will normally be values. Unless they need to be referentially shared, relations included in the state of other ADTs will also normally be values (e.g., a project object might...
include in its state a value that is a relation between task and employee objects).

Unlike relations (which are ADTs and may be either values or objects) the tuples within a relation are not ADT instances, and, therefore, cannot be objects. The only access to a tuple in a relation is thus through operations provided by the containing relation. This is necessary to correctly enforce the encapsulation required by the semantics of relations. For example, allowing a tuple in a relation to be an object would allow it’s attributes to be changed without knowledge of the encapsulating relation. In contrast, the relational model requires a tuple “update” to be performed using operations on the relation -- a delete followed by an insert. Operationally, this is important for supporting consistency constraints.

Tuple attributes themselves are either primitive atomic data or ADT instances (which may be objects). Value attributes simply provide functionality consistent with current relational systems supporting ADTs; alternatively, object attributes allow relations that are based on identity. For example, a father/daughter relationship is based on identity and doesn’t change if the daughter marries, changes her name (i.e., her state), or even, possibly, her behavior (i.e., her ADT -- see the following discussion).

**OOP in the Model**

What is normally (in object-oriented terminology) called a class is called an ADT in the model discussed here. ADTs are ordered in a behavioral hierarchy that represents inheritance of abstract interface -- not implementation\(^1\). The model does not restrict ADT implementations, so inheritance of implementation (as in class-based object-oriented programming) is allowed but not required.

The model is unique in that it allows exceptions to behavioral inheritance while retaining static type safety. Benefits arising from this capability include the ability to allow induced inheritance for parametric ADTs while retaining type safety, and the ability to statically typecheck object specializations, in which an object changes its ADT (thereby, in general, replacing its state) in order to change existing behavior, or acquire new behavior while retaining its identity. To the author’s knowledge, no other existing data model provides these capabilities (both of which are especially important in a database context).

OODBs often make use of class extents, which provide a useful indexing technique for objects. In the model discussed here, extents appear as instances of ADTs whose state includes a set of objects plus a set of sub-extents. Operations on an extent allow adding or removing objects, and adding or removing sub-extents. The primary operation on an extent returns a set containing all its contained objects (including those of its sub-extents). Databases and languages based on the model may provide certain extents automatically, as part of a global execution environment.

**Unifying the Underlying Technologies**

Problems and possibilities for technological integration will be discussed during the panel presentation.

**References**


\(^1\) See [Har91] for an interesting discussion concerning the desirability of this separation.