Towards a Spatial-Temporal Processing Model

Jonathan B. Lori

Abstract—**This paper discusses architecture for creating systems that need to express complex models of real world entities, especially those that exist in hierarchical and composite structures. These models need to be persisted, typically in a database system. The models also have a strong orthogonal requirement to support representation and reasoning over time.**

***Index Terms*—Spatial-temporal processing, object-relational mapping, entity-relationship modeling, design patterns, dynamic composites**

I. Introduction

SINCE using relational databases and object oriented programming (OOP) languages have become commonplace for developers, it is only natural that systems have evolved to facilitate using relational databases as data persistence mechanisms for programs developed in object oriented languages. For example, the Java Database Connectivity (JDBC) API [1] provides database-independent connectivity between the Java programming language and a wide range of databases.

Object-based systems are founded on a set of fundamental concepts [2]. Objects have state, so they can model memory. They have behavior, so that they can model dynamic processes. And they are encapsulated, so that they can hide complexity. There are only two kinds of relationships in an object model [3], a static relationship: inheritance ('is-a") and a dynamic relationship: composition ('has-a").

As OOP has advanced, other structuring facilities have emerged in designs and code based on idioms and best practices that have evolved in OOP-based systems. Some of these practices have been codified as "Design Patterns" [4]. One such object oriented design pattern is **Composite**. This pattern composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Evolution was also occurring in the database world. Although initially discounted by the relational community at large, the ER model [5] is based on strong mathematical foundations, including: Set Theory, Mathematical Relations, Algebra, Logic and Lattice Theory.

Manuscript received October 10, 2009.

J. Lori is a PhD candidate at the University of Bridgeport, Bridgeport, CT, 06066 USA (email: jlori@bridgeport.edu).

This work was partially funded by the Pratt and Whitney Corporation, 400 Main Street, East Hartford, CT 06453 USA.

At the beginning of this decade, Dr. Ralph Kimball, one of the leading visionaries in the architecture of Data Warehouse systems described his goals for the marriage of database technology, ER models, and object oriented programming systems. In his newsletter [6], Kimball proposes four kinds of data warehouse business rules: "These rules included simple data formats, relationships between the keys of connected tables, declarations of entity relationships, and 'complex business logic'…" Kimball wanted direct support in the programming system for the third rule, particularly in the situation where many-to-many relationships were used.

Describing the fourth rule, Kimball states: "Complex business logic will always remain a combination of static data relationships and adherence to procedural sequences…"

Dr. Kimball sought an approach that uses OOP to manage entity-relationship data models and implements the associated processing logic to form an effective basis for data warehouses. While both OOP and Data Warehouse design had matured, a major stumbling block remained to be overcome. The problem is known as "object-relational impedance mismatch". Ambler [7] supplies this definition, which focuses on the orthogonal approaches to search and navigation in the two models: "The object/relational impedance mismatch is the difference resulting from the fact that relational theory is based on relationships between tuples that are queried, whereas the object paradigm is based on relationships between objects that are traversed."

As software technology moved forward through the first decade of the twenty-first century, a new technology emerged for integrating OOP and database systems. This technology is known as Object-Relational Mapping (ORM). ORM is defined as follows [8]: "Object-relational mapping (ORM, O/RM, and O/R mapping) in computer software is a programming technique for converting data between incompatible type systems in relational databases and object-oriented programming languages. This creates, in effect, a 'virtual object database' that can be used from within the programming language."

By the middle of the decade, ORM systems became highly sophisticated and had achieved significant results. Some of the best ORM implementations are open source Java-based systems [9]. These systems brought back a lightweight, object-oriented persistence model based on the concept of POJOs (Plain Old Java Objects) [9].

II. Problem Space

The architecture discussed here is realized in a system called Phoenix [10]. The system is designed to implement a management suite for jet engines. The heart of the suite is an

T. Sobh, K. Elleithy (eds.), *Innovations in Computing Sciences and Software Engineering*,
DOI 10.1007/978-90-481-9112-3_20,

application known as On Wing Tracker (OWT). The purpose of OWT is to track the configuration and utilization of engines, engine components and parts. Essentially, this is a classic Bill of Materials (BOM) problem. However, there are a few other critical elements to the problem. Engines are complex and expensive assemblies that last for decades. Engines evolve. Components wear out. Modules and parts are moved from one engine to another. Information on the state of the engine may be "late arriving" and sometimes missing.

Utilization may be expressed in multiple ways, from simply accumulating run time hours to more sophisticated event-based modes such as throttle operations per flight. What is required to solve such a set of problems is not simply a system structured around a spatial dimension i.e. a BOM model, but one which can also reason over temporal dimensions as well.

III. Design patterns

As stated earlier, object models contain two types of relationships: composition and inheritance. As Wirfs-Brock [3] points out: "Both (models) have analogs in a family tree. A composite relationship is like a marriage between objects. It is dynamic, it happens during the participating objects' lifetimes, and it can change. Objects can discard partners and get new partners to collaborate with. Inheritance relations are more like births into the family. Once it happens, it is forever… We can extend an object's capabilities by composing it from others. When it lacks the features that it needs to fulfill one of its responsibilities, we simply delegate the responsibility for the required information or action to one of the objects that the object holds onto. This is a very flexible scenario for extension."

When first considering a BOM model, which is essentially a tree structure, an architect may be tempted to begin modeling based on inheritance. However, an architecture organized around composition is a dynamic and flexible approach, and more extensible. There is a long-recognized design axiom [11] that states: "Prefer composition to inheritance."

The interesting point in the Phoenix architecture is that if there is one major organizing principle it is this: the system is organized around the notion of Dynamic Composites [10]. By this it is meant that BOM hierarchies are built as Composites, where a Composite, while already a dynamic OO relationship, is also assembled from a dynamic search. The search is through information stored in a generic ER model that is in turn stored in a relational database.

Phoenix is logically composed as a generic Entity-Relationship model that is persisted in a relational DBMS system. (Fig. 1) The generic ER model is then mirrored by a generic object model. (Fig. 2) The two models are mapped together through an object-relational mapping system. In the Phoenix architecture, Hibernate is the ORM [9]. The ER model is decimated enough to produce the desired flexibility, including the capability of "decorating" entities with any required attributes. Thus, the ER model provides a unified data model for the system. The object model is closely matched to the ER model. Therefore it is easy to fulfill all of Kimball's goals for using OOP to drive an ER model-based data warehouse.

Note also that there are no entities or classes called "Composite". This is because the dynamic composites exist only as sets of instances in memory. Finally, note that the entities (tables) and their mirrored classes contain strategically embedded timestamp fields. The object model contains both Java code (procedural logic) and embedded queries and parameters (SQL/HQL). (SQL is the Structured Query Language. HQL is the Hibernate Query Language [9]).

IV. Temporal Processing

Temporal reasoning [12] is handled as follows. A Bitemporal Database is implemented using the foundation provided by the Phoenix Architecture. Facts are stored in a database at a point in time. After the fact is stored, it can be retrieved. The time when a fact is stored in a database is the transaction time of the fact. Transaction times are consistent with the serial order of the transactions. The past cannot be changed; therefore transaction times cannot be changed. A transaction time also cannot be later than the current time. Typically, the commit time of a database transaction is used as the transaction time.

Conversely, the valid time of a fact is the time when such a fact is true in the modeled reality. A fact can be associated with any number of events and intervals. The system uses transactional storage mechanisms to persist data. Such a storage event corresponds to a transaction time for that event. Meanwhile, the data being stored also contains representations of a valid time event: "Something was done to an entity or a characteristic of an entity at some (valid) time". A transaction-time database supports transaction time and such a transaction can be rolled back to a previous state. A valid-time database contains the entire history of the entities it contains. Phoenix maintains and uses both the transaction time and the valid time information to provide temporal reasoning in Domain Models built using the Phoenix Architecture. Hence, the Bitemporal Database is a built from the combination of the Domain Model, the structuring of entities within the Dynamic Composites that comprise the model and the ability to track the history of each entity and its characteristics in an arbitrary fashion.

V. System Architecture

The Phoenix Architecture has implemented a novel approach for processing, tracking and calculating information when the representative structure, characteristics of the structure, and the temporal history of both the structure and the characteristics of its components may be easily tracked and modified over time. This includes the capability to re-materialize the representative state at some arbitrary point in time. The innovations are in three primary areas: