5-2005

Extending the Abstract Data Model.

Matthew Bryston Winegar
East Tennessee State University

Extending The Abstract Data Model

A thesis
presented to
the faculty of the Department of Computer Science
East Tennessee State University

In partial fulfillment
of the requirements for the degree
Master of Science in Computer Science

---

by
Matthew Bryston Winegar
May 2005

Donald B. Sanderson, Chair
Steven Jenkins
Phillip E. Pfeiffer, IV

Keywords: Data Modeling, Semantics, Schema Translation.

ABSTRACT

by

Matthew Winegar

The Abstract Data Model (ADM) was developed by Sanderson [19] to model and predict semantic loss in data translation between computer languages. In this work, the ADM was applied to eight languages that were not considered as part of the original work. Some of the languages were found to support semantic features, such as the restriction semantics for inheritance found in languages like XML Schemas and Java, which could not be represented in the ADM. A proposal was made to extend the ADM to support these semantic features, and the requirements and implications of implementing that proposal were considered.

CONTENTS

Page
ABSTRACT 2

Chapter

1. INTRODUCTION 6

1.1 – Data Modeling and Data Semantics 6
1.2 – Data Translation and Semantic Loss 6
1.3 – Overview of Results 7
1.4 – Overview of the Balance of this Thesis 8
2. BACKGROUND 9

2.1 – Database Modeling 9
2.2 – Semantic Database Modeling 9
2.2.1 – Codd's Relational Model 10
2.2.2 – E/R Modeling 11
2.2.3 – OSAM* 12
2.2.4 – Format Model 12
2.2.5 – IFO Model 13
2.3 – Translation as Migration 14
2.4 – ADM/ASM 15
2.5 – The ADM 16
3. LANGUAGES OVERVIEW 18

3.1 – Introduction 18
3.1.1 – Python 18
3.1.2 – Java 19
3.1.3 – PHP 20
3.1.4 – C# .NET 21
3.1.5 – VB.NET 22
3.1.6 – Oracle OO Extensions 22
3.1.7 – Prolog 23
3.1.8 – XML Schemas 24
4. ANALYSIS 26

4.1 – Introduction 26
4.1.1 – Python 26
4.1.2 – Java 26
4.1.3 – PHP 27
4.1.4 – C# .NET 27
4.1.5 – VB.NET 28
4.1.6 – Oracle OO Extensions 28
4.1.7 – Prolog 29
4.1.8 – XML Schema Definition Language (XSD) 29
4.2 - Findings 30
5. SUGGESTIONS 31

5.1 – ADM Taxonomy 31
5.2 – Extension to the Taxonomy 32
5.3 – New Focus for the ADM 33
6. IMPLEMENTATION 36
6.1 – ADM Changes 36
6.2 – ASM and Migrators 37
7. CONCLUSIONS 41
REFERENCES 43
VITA 45

CHAPTER 1

INTRODUCTION

1.1 – Data Modeling and Data Semantics

A data model can be defined as an abstract, logical definition of a system's components that characterizes the structure of the data. [6] In the context of databases, semantic data modeling is an attempt to capture and express the meaning of a database's data, without regard to the data's implementation. [4] The quality of a semantic data model is determined by how well that model expresses the intension of the data. This quality is also dependent on the model's level of abstraction. A model that tracks only inheritance hierarchies might be a limited, but good model, within its level of abstraction.

1.2 – Data Translation and Semantic Loss

The limitations of a model's ability to fully capture intension are often exposed in the context of data translation. Information that has been represented in one format must often be translated into other formats as a condition of exchanging that information. For example, people who speak different languages must often use translators to exchange speech and ideas. Translation is also a common need when sharing information stored in computers. These translations can be as simple as the conversion of a Windows text file into a UNIX text file—or as complex as the conversion of XML query into a format recognized by an SQL database.

Data translation, unfortunately, is not a trivial problem, due to the difficulty of ensuring that the translation preserves the meaning of the original information. The ability to preserve meaning is limited by the existence of relationships in the original format that cannot be expressed in the target format. The quality of the translation is also affected by the implementation of the methods used to perform the translation.

The quality of a particular scheme for data translation can be evaluated by modeling this translation as a function, or a composition of a sequence of functions, and characterizing the accuracy of the mapping that this composition provides. Pragmatic concerns related to how

these mapping(s) are defined impose important limitations on the quality of a translation. For example, translation schemes that allow any of N formats to be converted to any of these same N formats are typically structured as a two-stage conversion, involving

- a set of N translation functions that convert each of the N formats to an intermediate format, together with
- a second set of N translation functions that convert the intermediate format to each of the N original formats.

This scheme for data translation, which requires 2N functions, is substantially easier to implement than a set of N*(N-1) direct translations from every format to every other format. Unfortunately, the use of the intermediate format can also create data loss, if the intermediate format cannot represent some relationships in one or more of the other formats. Therefore, for this type of translation to be equivalent (in terms of theoretical minimum semantic information loss) to a single step translation function, the intermediate format must support all semantic features of the models to which the intermediate format applies.

1.3 – Overview of Results

The starting point for the work in this thesis, the Abstract Data Model [19], was devised by Sanderson and Spooner as a tool for tracking semantic information loss in translation. Sanderson used an Abstract Semantic Model (ASM) to model the semantic content of computer programming and data specification languages to track semantic information loss in translation. The ADM modeled the schemas being translated, and the ASM was used to track the semantic effects of that translation.

The ADM was developed by Sanderson and Spooner to represent schemas in Basic, C, C++, EXPRESS, Fortran, Hierarchical Model, Lisp, Network Model, Objective C, OSAM*, Pascal, PL/1, Pratt's General Model, Relational Model, Rose 2.3, Smalltalk-80, and SQL. The ADM was used in the context of modeling and predicting semantic loss in translation between these languages.

The main results of this thesis come from attempting to represent eight newer languages in the ADM: Python, Java, PHP, C# .NET, VB.NET, Prolog, Oracle OO Extensions, and XML Schemas. Four of the languages, Java, C# .NET, VB. NET, and XML Schemas, were found to support restrictions on inheritance that could not be represented in the ADM. This limitation was then addressed by a proposed extension to the ADM that captures the semantics of the additional, inheritance-related features in these four languages.

1.4 – Overview of the Balance of this Thesis

The balance of this thesis is divided into six chapters. Chapter 2 reviews previous works dealing with semantic modeling. Chapter 3 surveys the eight programming languages that constitute the starting point for this work: languages that were not used to develop the original ADM. Chapter 4 analyzes the degree to which these languages are supported by the ADM. Chapter 5 suggests extensions to the ADM that enable it to support those features of these eight languages that the ADM cannot now support. Chapter 6 describes what must be done to implement the proposed changes to the Abstract Data Model. Finally, Chapter 7 concludes with a brief recap of the results of this thesis, and proposals for further research.

CHAPTER 2

BACKGROUND

2.1 – Database Modeling

A database system is a system for storing and recovering information. A model of a database system can be viewed as a two-part entity: i.e., a model of database system operation, together with a model of the data that this system contains.

A database system is modeled as an "abstract machine" for storing and recovering information. [6] Models of database systems can be viewed as abstractions that specify a system's logical structure. [10] A common model has four logical components: a data space, which consists of elements and the relationships between them; type definition constraints, which define valid and invalid relationships within the data space; manipulation operations, which allow modifications to the database; and a predicate language, which allows one to select elements of the database based on a comparison to properties of those elements. [10]

A data model for a database system defines how the database's users can structure the database's content. From a practical point of view, the most important data model for contemporary databases is the relational model. This model structures data in a database as a set of tables, and CRUD (create/ read/ update/ delete) operations on these tables as queries that manipulate subsets of the information in the whole database. [6]

2.2 – Semantic Database Modeling

A semantic model is an abstraction that attempts to capture the meaning of some real-world phenomenon. Semantic models, by dealing with meaning, relate to real-world relationships that exist between objects that the database elements model.

Semantic models for information in databases attempt to model the meaning of a database's data [10]. According to C. J. Date, "Semantic modeling ideas can be useful as an aid in database design, even in the absence of direct DBMS support for those ideas." [6] Semantic database modeling can provide checks and guidelines for implementing a database in a way that

is consistent with the semantic meaning of the data being represented. Practitioners routinely design models that focus on certain aspects of the data being modeled, and ignore others, in order to make key aspects of a dataset easier to understand.

Semantic data models have been developed by extending existing database models with semantic elements. A very basic extension to a database system would allow the database to "know" the "type" of different fields, and to detect invalid operations based on the types of the fields. This would be analogous to data typing in programming languages.

The next section reviews five semantic models: Relational Model; E/R modeling; OSAM; the Format Model; and the IFO Model Most of these models were concerned with a set of common issues related to the modeling of data: aggregation, classification, and supertype-subtype relationships. This commonality reflects shared concerns with modeling real concepts, for which naturally occurring relationships were common.

2.2.1 – Codd's Relational Model

Two key abstractions supported by Codd's Relational Model are aggregations and generalizations. [20] An aggregation is an abstract object that captures a relationship between two or more objects. A generalization is an abstraction object that represents a set of related objects. Aggregations and generalizations allow a modeler to emphasize certain aspects of some data while ignoring other, irrelevant aspects of the data.

Smith identifies three benefits of incorporating aggregations, generalizations, and abstraction hierarchies into relational schema. First, these constructs support the creation and use of different views, or abstractions, of a database system. This capability, which allows the modeler to provide different users with different abstractions of the data, is similar to the support for views provided by standard SQL. Second, the discipline afforded by these constructs helps to insulate models from minor changes to a dataset's schema. The abstraction hierarchy will tend to encapsulate the changes to the level of the hierarchy where they occur, and higher levels will not be affected. Finally, these constructs support the development of self-documenting models, by allowing a model to be structured as a progressively detailed set of refinements of an

initial, simpler model. Smith states that aggregation has been a primary focus of database research, while artificial intelligence research has been concerned with generalization in knowledge bases. [20]

2.2.2 – E/R Modeling

One well-known model for semantic information, the entity/relationship (E/R) model, supports four basic constructs for modeling data: entity, relationship, property, and subtype. [6] Entities are similar to objects in object-oriented design; they represent or model some 'thing' related to a problem space. Relationships, entities that model relationships between two or more other entities, are analogous to Smith's aggregation abstraction described above. Properties, which describe entities and relationships, are analogous to member data for an object. C. J. Date defines a subtype as follows: "Entity type *Y* is a subtype of entity type *X* if and only if every *Y* is necessarily an *X*." [6] Subtyping is analogous to the 'is a' relationship commonly encountered in object-oriented design, and is closely related to the generalization abstraction presented by Smith.

Additional refinements to the E/R model provide finer classifications of entities, relationships, and properties. The model further classifies entities as either *weak entities*, those whose existence depends on another entity, or as *regular* or *strong entities*, those that exist independently of other entities [6]. Similarly, a type T's participation in a relationship can be classified as *total* or *partial*, according to whether all entities of type *E* participate in a relationship *R* [6]. This concept of total participation and partial participation is closely related to the concept of weak and strong entities, because a weak entity necessarily has a relationship to another entity upon which its existence depends—and must therefore participate totally in that relationship. Properties describe the entities or relationships to which they apply. Properties may be simple data primitives or a composite data structure, and they may be single or multi-valued. [6]

The E/R model is associated with the Entity/Relationship Diagram. An E/R Diagram represents information in an E/R model, such as entities and their relationships, in a visual

format. In an E/R Diagram, entities are represented by labeled rectangles, and labeled connecting lines model the relationships between those entities. [15] Each line of an E/R Diagram's relationship lines also characterizes a relationship's cardinality, or the number of entities that may be on each side of the relationship line, and its modality, or the need for entities to be present on one of the sides in order for the relationship to exist. [15]

2.2.3 – OSAM*

The Object-oriented Semantic Association Model (OSAM*), is one of several extensions to database models that add semantic capabilities to a database. OSAM* uses object-oriented techniques to define the semantics of data. [21] An Object-oriented Query Language associated with OSAM* supports association and pattern type queries instead of traditional attribute queries. [21]

2.2.4 – Format Model

The Format Model supports three recursive constructors: composition, classification, and collection. [8] The composition constructor, which creates a new format from other existing formats, is closely related to Codd's aggregation abstraction [8, 20]. The composition constructor also appears to be analogous to relationships in E/R models. The classification constructor, which assigns a higher-level type to another format, is comparable to subtyping in E/R models and generalization in Codd. It differs from generalization of Codd, however, in that "an instance of a classification format is (essentially) a single member of the domain of one of the underlying formats", but "an instance of a generalization is a set of objects, each belonging to (at least) one of the underlying domains". [8] The collection constructor defines sets of objects, all of which have the same type. [8] Interestingly, in [8], the claim is made that the generalization concept from [20] can be modeled "as a collection of a classification". This would indicate that the Format Model allows more fine-grained specification of semantic information.

2.2.5 – IFO Model

Abiteboul and Hull's IFO model [1] is a formal, graphical database semantics model. The authors claim that the model supports four basic precepts of semantic database modeling: it models information regarding objects and their relationships in a direct manner that captures the problem space that the data represents, instead of the details of how those relationships are implemented; it emphasizes the use of functional relationships in databases; it supports the use of ISA relationships in semantic data models in order to identify an object as a type of some more generic object; and, finally, it supports the hierarchical construction of objects from other objects. [1] Abiteboul et al. claim that IFO subsumes several other semantic models, which is not surprising, given that semantic databases tend to model the same kinds of real-world relationships.

The IFO has several types of elements to graphically model the semantics of a database schema. The IFO model supports three atomic types and two type constructors [1]. The first atomic type, printable, includes pieces of data that can be input or output, like strings and other primitives, and (even) sound files. [1] The second type, abstract, includes objects that are being implemented or modeled in a database that correspond to some real-world objects with no underlying structure. [1] The final atomic type, free, includes entities obtained via the ISA relationship. The first type constructor, "star-vertex", corresponds to the collection of the Format Model [8]; it indicates a grouping of different objects. [1] The other, "cross-vertex", corresponds to the aggregation abstraction of [20] and composition in [8]. [1]

The IFO model allows representation of functional relationships by using fragments. [1] Graphically, fragments are just labeled arrows that point from one element in the graph to another. The label indicates what the fragment is modeling.

The IFO's model's final components are two elements that support two different kinds of the ISA relationship: specialization and generalization. [1] Specialization models the notion of roles; specifically, the notion that an object's role (for example, a person could sometimes be a student and sometimes be a teacher) can change. [1] Generalization models the notion of a generic virtual type that cannot be directly instantiated, but that can be used to define other

derived types. [1] The classic example of generalization is shape, which is a generic type for such objects as triangle, circle, and rectangle.

All of the four elements of the IFO can be combined to model complex systems. Abiteboul and Hull refer to models built by combining these elements as schemas. The schemas should be built top-down, as there are certain contradictions that can arise from incorrect use of the ISA relations in the diagrams. [1]

2.3 – Translation as Migration

The preferred strategy for N-way data translation described in Section 1.2 uses an intermediate data format (IDF) to support translation. Since translations between data formats can lose semantic information, this IDF should ideally support any information that can be represented in any of the original N formats. This constraint on the IDF can be expressed by relating it to the best possible direct translation, X_1, from a source format *A* to a target format *B*. If the amount of information lost by the best possible two-stage translation from source, to intermediate, to target format is equal to the amount of information lost by translation X_1, then the IDF can be considered a *neutral format* with respect to *A* and *B*.

Sanderson showed that a two-step translation could be implemented by using a DBMS as an IDF. Specifically, Sanderson used a DBMS to store an analog of the source format, and migration functions to translate the stored source format into an analogue of the target format, which could then be 'read' into the target format directly. [19]

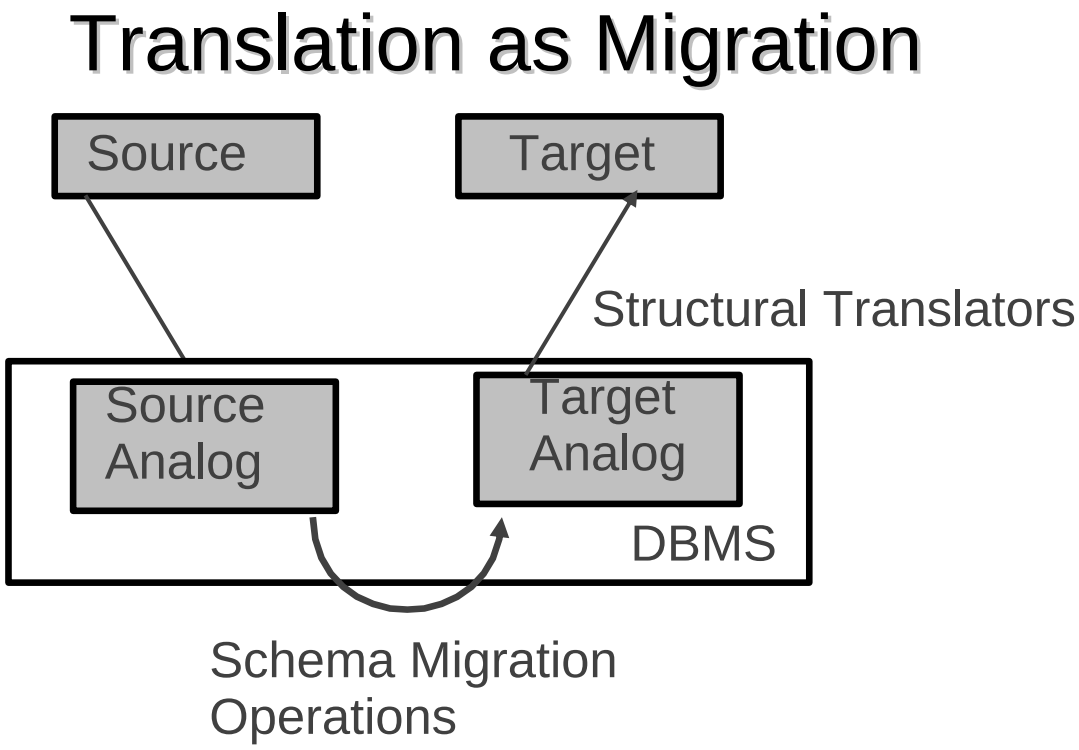

Figure 1- A technique for translating from a source schema to a target schema.

This "Translation as Migration" model (see Sanderson and Spooner [19] for more detail) for translating between data formats simplifies the problem of analyzing a translation by realizing the bulk of the transformations in a single format. The migration functions can be defined for the neutral database format such that the semantic effects of the migration functions are understood and modeled. Thus, it is feasible to model and predict semantic change in translations between languages supported by the neutral format.

2.4 – ADM/ASM

Sanderson's Abstract Data Model (ADM) is a 'metamodel' of data modeling languages that could serve as a neutral format for the 'Translation as Migration' strategy for data translation (cf. §2.4). [19] The Abstract Semantic Model (ASM) is a model that was used by Sanderson to track the semantic effects of migration operators on schemas represented in the ADM. [19] Since the ASM is used to predict semantic data loss in translation, it needs to store information

about the system being modeled, as well as information about what kinds of semantic information are supported by the data modeling languages involved and information about the schema being used. [19] The ADM, when combined with the ASM, can model the semantic information of the modeling languages supported by the ADM, the schema being used, and the system being modeled. This allows systematic predictions about semantic loss that occurs as a result of the migration functions being used to translate a model of a system. [19]

2.5 – The ADM

The Abstract Data Model (ADM), which is the main focus of this research, was originally developed to support semantic features of Basic, C, C++, EXPRESS, Fortran, the Hierarchical Data Model, Lisp, Network Model, Objective C, OSAM*, Pascal, PL/1, Pratt's General Model, the Relational Data Model, Rose 2.3, Smalltalk-80, and SQL. The model, intuitively, is an object-oriented model that represents the static data structures supported by languages.

More formally, the ADM is defined as an eight-tuple, consisting of the following component elements:

S – (D, N, Δ) – The state S, defined in terms of domains D, names N, and defined data objects Δ. The state S is the representation of the static aspects of a schema in the ADM.

P – (D_0, N_0, Δ_0) – Initial state, consisting only of system defined domains D_0, names N_0, and defined data objects Δ_0 .

M – A set of data manipulators, which construct and modify data objects.

O – The space from which valid object IDs (Oids, used internally by the ADM) are drawn.

C – A set of constructors, which create complex domains using previously defined domains.

G – Grouping operators, which construct aggregates of existing domains.

A – The alphabet from which valid names are constructed.

I – Inheritance operations, which model the inheritance hierarchies that exist between domains.

In the categorization above, a domain refers to a type. The state S, then, contains a set of domains D that are the types present in a runtime, and defined data objects Δ that are of the domains D. The names N of the state S are mapped to the domains D such that each name corresponds to one domain. The data manipulators M define how data objects may be modified, and how they are created; they generally correspond to class methods in object oriented programming languages. Oids are programmer-accessible unique object identifiers. They are useful for tracking semantic change in data translation for several reasons. First, if an object is split into multiple objects during migration, or merged with another object, this can be traced by using an Oid history. Secondly, Oids are useful for modeling inheritance. If an object is of a class with some supertype, a list of 'superoids' can be maintained, and a superoid history can also be maintained, similar to oid histories, to track changes with respect to inheritance hierarchies. Oids, superoids, and inheritance were included in ADM to support object-oriented languages; they would not be needed to support structural programming languages like C.

The categories of the ADM of Constructors and Groupings may be further refined. In [19], Sanderson uses a [Degree, Heterogeneity, Construction] triple to categorize data structures. The Degree can be either Single or Multiple, indicating whether a data structure holds a single instance or multiple instances of a data item. Heterogeneity indicates what components make up each data item: possible values are Atomic, which indicates there is one instance of one type; Structure, which indicates one instance each of several types; and Union, which indicates one item is chosen from a list of possible types. Finally, Construction can be either Simple, which indicates that the components of the data item consist only of system-defined domains such as Integer or Char, or Recursive, which indicates that the components of a data item may consist of user-defined types.

CHAPTER 3

LANGUAGES OVERVIEW

3.1 – Introduction

The following section presents key features of the languages featured in this study: Python, Java, PHP, C# .NET, VB.NET, Oracle OO Extensions, Prolog, and XML Schemas.

3.1.1 – Python

Python [16] is a high level, interpreted programming language that supports standard object-oriented programming constructs, including classes, single and multiple inheritance, and virtual functions. Private data protection for classes is only by convention; it is not enforced. Instance objects of a class only understand attribute references, which can be either data attributes or methods. Python classes may have static data, which persist across all instances of the class.

Python supports runtime type-checking. This type-checking is automatic only for built-in operators and primitive constructs, like arithmetic and logical operators and statement guards. All other type-checking must be hand-coded, using primitives like Python's isinstance(). This lack of automated type-checking contrasts with typing in compiled languages like C++, which generally require the use of type declarations.

Functions in Python may accept either a set arguments, some of which may have default values, or a variable number of arguments. Functions may also accept named parameters, in the form of lists that bind keywords to input values. Functions either return a value(s) or have no return value. Python also supports 'lambda' functions, which allows convenient anonymous functions to be written to accomplish some tasks.

Python's set of built in types include strings (str), lists, tuples, and dictionaries (dict): maps that pair keys with values, including other dictionaries. All of these types are subclassable, indexable, and support built-in operations. They are also either mutable or immutable. In Python, a mutable type is a type whose component values can be modified in place, whereas

immutable types cannot. Strings are immutable, lists are mutable, tuples immutable, and dictionaries mutable. An immutable type may have a mutable element; for example, one element of a tuple may be a list, and that list may be modified, but it may not be replaced with something else directly. Dictionaries in Python are one-way associative arrays.

Python supports integers, floating point numbers, complex numbers, booleans, and "NoneType" as primitive types. A special value of type NoneType, 'None', denotes nothing, or no value.

3.1.2 – Java

Java [7] is a high level compiled Object Oriented programming language. Java was intended to be a fully cross-platform language, in that programs written and tested on one platform would compile and run on any other platform. This goal is accomplished, in theory, by having Java programs compiled and run via a Java Virtual Machine.

Java supports single inheritance for classes and interfaces, instead of multiple inheritance for classes. Java supports public, private, and protected attributes and methods for classes, enforcing them at compile time. Classes may have static data in Java. Java supports abstract classes, classes that are intended only for inheritance and contain method declarations that must be defined in subclasses that will be used.

Methods (member functions) in Java take zero or more parameters and can return a value or no value. Most methods automatically take an implicit parameter referring to the current object. Methods that do not take an implicit parameter are 'static' methods that can be called without having an instance of the class that contains the static method.

Commonly used data structures in Java include arrays, strings, and vectors. A vector is a collection of objects that can grow dynamically. Arrays are collections that are declared with fixed type and size.

Java supports the following primitive types: int, double, and bool. The language also has a special value, 'null', which can be assigned to an object name, and refers to 'no object'.

3.1.3 – PHP

PHP [14] is an Object Oriented interpreted programming language that is commonly used in web-based applications. PHP, a recursive acronym, stands for **P**HP: **H**ypertext **P**reprocessor. PHP supports access to a large number of relational databases, and has been ported to most major operating systems. [13]

PHP provides support for single inheritance and interfaces. Although PHP is an interpreted language, it does enforce the use of public, protected, and private qualifiers for constraining the visibility of data and methods. PHP supports abstract classes and static class members or methods. An instance of a class is called an object in PHP.

Functions in PHP support passing parameters by value or by reference. They support variable length argument lists and default values for parameters. Functions in PHP can return values, including objects or lists. Recent versions of PHP allow functions to be called before they are defined in the code, as long as the function is defined somewhere. However, if the definition of a function occurs inside a conditional block, then that function does not exist until the conditional block is entered.

In PHP, arrays consist of a sequence of key-value pairs, a data structure that is often called a dictionary. PHP supports two restricted types of ‘unions’: mixed and number. Mixed indicates that a variable may contain values of two or more types. Number denotes a variable that may refer to either an integer or a float. PHP also supports a callback type. A callback type is a name of a function that is passed as a parameter to another function so the first function can be used. In PHP, these three types, mixed, number, and callback are referred to as pseudotypes.

PHP supports the scalar types boolean, integer, float, and string. A PHP string is a sequence of single byte characters. The type resource refers to some resource granted to the program by the operating system, such as a file handle. Most resources are freed automatically by PHP's garbage collector when they are no longer needed. The value NULL is the only possible value of type NULL. NULL is the value of any variable that has not been set to any other value.

3.1.4 – C# .NET

C# .NET is an Object Oriented programming language that is part of the .NET framework. C# .NET, like other .NET family languages, is defined in terms of a virtual machine known as the Common Language Runtime, or CLR [5,23]. Libraries, modules, and classes developed in one language supported by the .NET framework can be used in programs written in other languages supported by the framework. For this to work, all public interfaces of the libraries, modules, or classes must use only data types supported by all of the languages of the .NET framework, although the implementation details do not have to adhere to this standard (Common Language Specification). [23] It is even possible to derive classes in one language from base classes written in another language, provided that both languages are part of the .NET framework and the base class adheres to the Common Language Specification.

C# .NET supports single inheritance and interfaces for classes; it does not support multiple inheritance. For access control of member functions and data members, C# .NET supports public, protected, private, and internal. Internal indicates that a data member or function is accessible to all classes that are part of the same assembly, but not others. Abstract classes and member functions are supported, as are static data members and member functions. While abstract classes force implementation of certain functions in derived classes, C# also supports a concept called a sealed class, which is a class that can not be used as a base class for deriving further classes.

In addition to classes, C# .NET supports structs. Structs are a value type in C#, while a class is a reference type. For collections, C# supports arrays that derive from System.Array, which provides several methods for working with the arrays. Otherwise, they operate like traditional C or C++ arrays. Strings are also a native type supported by C#. They are associated with the type System.String from the .NET library, which provides methods for string manipulation.

C# .NET's set of built-in types includes byte, short, int, long, float, double, decimal (fixed precision), char, and bool. The types byte, short, int, and long can be either signed or

unsigned. In C# .NET, null is a keyword that indicates a null reference, or a reference that refers to no object. The default value for new reference variables is null.

3.1.5 – VB.NET

VB.NET [2] is an Object Oriented programming language that is part of the .NET framework. VB.NET is built on the .NET CLR, like C# .NET. VB.NET is a completely different language than VB 6. Compatibility with older Visual Basic versions was broken to make VB.NET work with the CLR.

VB.NET supports single inheritance and interfaces. Accessibility options for classes include public, private, and protected; friend, which indicates that the member variable or function is accessible to any class in the same executable, dll, or assembly; and protected friend. Static variables for classes and functions are supported.

In VB.NET, an instance of a class is called a Reference Object. Reference Objects are allocated on the heap. Structures, which are stack-allocated value objects, correspond to C/C++ structs. Strings in VB.NET are variable length, as are arrays.

Primitive types supported by VB.NET include Byte, Char, Boolean, Decimal, Double, Short, Integer, Long, and Single. Decimal in VB.NET is not the same as decimal in C# .NET; rather, it replaces the old Currency type of VB 6. VB.NET has no support for unsigned types. Although unsigned types are part of the CLR, they are not included in the CLS, so VB.NET works with modules from other .NET languages that are CLS compliant.

3.1.6 – Oracle OO Extensions

Oracle OO Extensions for Oracle 9i are intended to expand the capabilities of Oracle's SQL and provide some of the benefits of Object Oriented programming to database schemas. Oracle OO Extensions support the structural and semantic features of standard SQL, together with the additional features that follow. [17]

Column objects are a user-defined type that can be used as an attribute in a relational table. They are essentially an Abstract Data Type, containing data that represents the state of the

object, and possibly methods that govern the changes that occur to the object. The methods that are defined for a column object can be implemented in PL/SQL, C++, or Java.

Row types allow a table to be defined where each row of the table is an instance of a column type. Each row then has a unique object ID. This type of table is an object table.

VARRAY is an object type to represent a collection of objects or other base data types. When declaring a VARRAY type it is necessary to specify a maximum size for the type. The space for the collection of objects is allocated inline in the table, so a maximum must be specified.

A nested table is a type that is a table. Because the type is a table, it can represent a collection of objects or other base types that is of variable size, because a table can have a variable number of records. This also means that the data of a nested table type exists out of line, in contrast to VARRAY type, where the data exist inline.

3.1.7 – Prolog

Prolog [18] is a declarative programming language. Prolog's model of computation differs from the traditional imperative model embodied in languages like Java. In imperative programming languages, programs are structured as a sequence of statements to be executed. In contrast, a Prolog program is structured as a set of facts (factbase) and a set of rules for deriving new facts (assertions). A Prolog program is executed by specifying a proposition (query) relative to a factbase and set of assertions. The Prolog interpreter then determines whether a fact that satisfies this proposition can be determined from these facts and assertions.

Trees are the primary data structures used in Prolog. Generally, when stating facts, the root of a tree represents a relationship and the leaves of the tree represent the participants in that relationship. Prolog supports the creation of complex data types using the same tree structures. For example, one could have a fact that states that an employee has a name, a social security number, and a number of dependents. Prolog supports lists and recursive definition, so a 'procedure' can be written to search for an element in a list by stating multiple definitions of a

deduction rule. If one definition of a deduction rule fails to return a fact, then Prolog will try the next one.

The primitives supported by Prolog are quite limited. In some versions only positive integers are supported. Very basic arithmetic operators and comparison operators are supported. Prolog enforces a single assignment paradigm for variables.

3.1.8 – XML Schemas

XML, the Extensible Markup Language, is a W3C language for representing documents. XML's salient characteristic is its support for user-defined grammars, in the form of mechanisms for constraining a document's form and content, over and above the minimal set of constraints imposed by XML proper. Currently, the most important of these mechanisms is the XML Schema Definition Language, or XSD [3,22], The XSD Language allows a user to define an object, an XML Schema, which, when referenced in an XML-standard way from within an XML document, provides a basis for validating that document's form and content.

XML Schemas are themselves XML documents. The root element of a schema is the <xs:schema> element. This element may indicate a namespace, which may even be a URL, of elements and data types that are used in the schema.

The XSD standard defines two primary constructors for type construction. The first, xs:simpleType, defines types that are similar in form and content to primitive types in other languages. A simple type is an element type that can only contain text or numeric data, i.e., it does not aggregate other elements and does not have attributes. The xs:simpleType declarations support the derivation of new simple types from 44 built-in XML types, including xs:string that serves as a sort of type of all simple types. The other primary type constructor, xs:complexType, defines types for objects that support subobjects, in the form of attributes and/or child elements. The precise form and content of complex types is specified using indicators: XSD elements that specify properties of aggregated elements like the order of elements, numbers of elements, and groupings to define related sets of elements. A schema's elements may be made extensible by

including the 'any' element in aggregations of elements, which acts as a wild card element. Similarly, for attributes, there is an 'anyAttribute' attribute.

The XSD standard also supports two constructs that support the derivation of subtypes from built-in and user-defined types. The one, the xs:restriction element, creates a new type by subsetting the space of values that its base type may assume. The other, the xs:extension element, creates a new type from complexType by adding fields to that complex type. The semantics of the restriction and extension elements are constrained in ways that allow any instance of a subtype to be substituted for an instance of its base type in any context in which that base type is allowed to appear.

CHAPTER 4

ANALYSIS

4.1 – Introduction

In this section the languages surveyed in the previous section were categorized according to the Abstract Data Model. Some of the languages were supported entirely by the model, but others supported semantic features that have been developed since the ADM was created. Thus, the ADM developed by Sanderson [19] did not support these semantic features. Generally, these unsupported features included restrictions on datatypes, such as the C# concept of sealed classes.

4.1.1 – Python

All of Python's constructs for modeling data are supported by the original ADM. These include:

Constructors- Structure { Single, Structure, Recursive },
Class-Subclass { Single/Multiple, Atomic/Structure, Recursive }

Groupings- Tuple, List, Dict, Collection Classes { Multiple, Atomic, Recursive }

Inheritance- single and multiple.

Primitives- int, float, boolean, str, complex number

Alphabet- (a-z, A-Z, 0-9, _)

Oids- Implemented using a hash for immutable objects, and using scoped names for mutable objects.

4.1.2 – Java

Most of Java's constructs for modeling data are supported by the original ADM. These include:

Constructors- Class-Subclass { Single/Multiple, Atomic/Structure, Recursive }

Groupings- Array, Vector, collection classes { Multiple, Atomic, Recursive }

Inheritance- Class-Superclass, only single inheritance with interfaces.

Primitives- int, double, bool, char, byte

Alphabet- (a-z, A-Z, 0-9, _)

Oids- Java has an Oid class, which represents Universal Object Identifiers, a dot separated sequence of positive integers. [11]

Java also supports 'final' classes, classes for which derivation of further classes is not allowed. This semantic feature is not supported by the original ADM.

4.1.3 – PHP

All of PHP's constructs for modeling data are supported by the original ADM. These include:

Constructors- Class-Subclass { Single/Multiple, Atomic/Structure, Recursive }

Groupings- array (really a dictionary), collection classes { Multiple, Atomic, Recursive }

Inheritance- Class-Superclass, only single inheritance with interfaces

Primitives- integer, float, string (sequence of byte-sized characters)

Alphabet- (a-z, A-Z, 0-9, _)

Oids- N/A

4.1.4 – C# .NET

Most of C# .NET's constructs for modeling data are supported by the original ADM. These include:

Constructors- Structure { Single, Structure, Recursive },
Class-Subclass { Single/Multiple, Atomic/Structure, Recursive }

Groupings- array, collection classes { Multiple, Atomic, Recursive }

Inheritance- Class-Superclass, only single inheritance with interfaces ('sealed' classes don't fit)

Primitives- byte, short, int, long, float, double, decimal, char, bool (signed and unsigned where it makes sense)

Alphabet- (a-z, A-Z, 0-9, _)

Oids- .NET has a Guid class that represents a globally unique identifier, a 128-bit integer.

C# .NET supports the concept of 'sealed' classes, which is analogous to 'final' classes in Java and not supported by the original ADM.

4.1.5 – VB.NET

Most of C# .NET's constructs for modeling data are supported by the original ADM. These include:

Constructors- Structure { Single, Structure, Recursive },

Class-Subclass { Single/Multiple, Atomic/Structure, Recursive }

Groupings- array (variable size), collection classes { Multiple, Atomic, Recursive }

Inheritance- Class-Superclass, only single inheritance with interfaces (NonInheritable and NonOverridable equiv of C# 'sealed') [9]

Primitives- Byte, Short, Integer, Long, Double, Decimal, Char, Boolean

Alphabet- (a-z, A-Z, 0-9, _)

Oids- .NET has a Guid class that represents a globally unique identifier, a 128-bit integer.

VB.NET supports semantic features analogous to C#.NET's 'sealed' classes, 'NotInheritable' and 'NotOverridable'. As has been mentioned in previous sections, this semantic feature is not denotable in the original ADM.

4.1.6 – Oracle OO Extensions

All of the Oracle OO Extensions for Oracle 9i are supported by the original ADM. These include:

Constructors- row { Multiple, Atomic/Structure, Recursive },

VARRAY { Multiple, Atomic/Structure, Recursive },

Nested Table { Multiple, Atomic/Structure, Recursive },

Type { Single, Atomic/Structure, Simple/Recursive }

Groupings- VARRAY, Nested Table

Inheritance- N/A

Primitives- N/A, or as per Oracle SQL

Alphabet- (a-z, A-Z, 0-9)

Oids- REF, logical pointer to a row object, replaces need for foreign keys [12]

4.1.7 – Prolog

All of Prolog's constructs for modeling data are supported by the original ADM. These include:

Constructors- fact (a tree) { Single, Atomic/Structure/Union, Recursive }

Groupings- list

Inheritance- None

Primitives- positive integers

Alphabet- (a-z, A-Z, 0-9)

Oids- N/A

4.1.8 – XML Schema Definition Language (XSD)

Most XSD constructs for modeling data are supported by the original ADM. These include:

Constructors- simple element { Single, Atomic, Simple },

complex element { Single, Atomic/Structure/Union, Recursive }

Groupings- complex elements that aggregate other elements.

Inheritance- base/derived data types: X is derived from Y if X either extends or restricts Y, also support for 'final'.

Primitives- xs:integer, xs:string, xs:double, xs:boolean, xs:date, xs:time

Alphabet- (a-z, A-Z, 0-9, _)

Oids- N/A

The XSD standard also supports several semantic features that are not supported in the original ADM. In addition to having support for a 'final' attribute, analogous to 'final' in Java, XML Schemas uses a more generic definition for inheritance than traditional languages. Specifically, in XML Schemas, a type X may derive from Y by either extending or restricting Y. If the type X derives from Y by restricting Y, then instances of X are substitutable in places where an instance of Y is called for. The restricting part is not supported in the current ADM.

4.2 - Findings

Two new semantic features were found that were not supported by the ADM. The first was the concept of a class (or type) that cannot serve as a base class for any other class. This semantic feature had to be supported in the ADM if the ADM were to be useful in predicting semantic information loss in translations involving languages that support such semantics.

The second semantic feature identified that was not supported seemed to be much more fundamental. It was found in XML Schemas, which employs a different definition for inheritance. In [19], the following characterization of inheritance was used for the ADM:

> If two domains A and B participate in an inheritance relationship, and B is a subtype of A (A is a supertype of B), then B contains as attributes all attributes of A.

Although this generic characterization of inheritance is adequate for most traditional languages that support inheritance, it is not sufficient for inheritance in XML Schemas. In traditional inheritance, if type B is a subtype of type A, then B can be said to extend A. For XML Schemas, if type B is a subtype of type A, then B either extends or restricts A. Also, inheritance in XML Schemas guarantees that the restricted subtype is substitutable for its supertype. The characterization given above is inadequate if inheritance may involve such restrictions. Thus, the ADM had to be extended to account for this type of inheritance.

CHAPTER 5

SUGGESTIONS

In the preceding chapters, the semantic features supported by eight languages that were not considered in Sanderson [19] were catalogued. Some of the semantic features in these eight languages are not supported by the ADM. These features include keywords to prevent a class from being used as a base class for another class, such as the Java 'final' keyword, and support for inheritance where the subtype either extends or restricts the supertype, rather than just extending the supertype. Both of these new semantic features were categorized as types of restrictions on reuse that should be tracked in the taxonomy of the ADM.

5.1 – ADM Taxonomy

In Sanderson [19], a taxonomy is presented that refines the Constructor and Grouping elements of the ADM. The taxonomy classifies data structures according to a triple consisting of degree, heterogeneity, and construction. The degree can be either Single, which indicates one instance of a data item, or Multiple, which indicates there are multiple instances. Heterogeneity can be either Atomic, indicating the data structure contains one instance of one type, Structure, indicating one instance of each of several types, or Union, indicating one type from a list of possible types. Finally, construction can be either simple or recursive. Simple indicates the data structure consists of only system-defined domains such as Integer, while Recursive indicates that the data structure may contain user-defined domains. Here are a few examples of C data types and the triple that would describe them in this model:

Union	{ *Single*, *Union*, *Recursive* }
Structure	{ *Single*, *Structure*, *Recursive* }
Array	{ *Multiple*, *Atomic*, *Recursive* }

The taxonomy allows the composition of data structures to be defined more precisely than the Constructor and Grouping elements of the ADM would allow. Data types in different languages that have the same triple according to their placement in the taxonomy are semantically similar with respect to degree, heterogeneity, and construction. By asking questions about where a data

structure fits in this taxonomy, one can recognize when data structures of the same name in different languages are actually different, or when data structures of different names are the same with respect to the model.

5.2 – Extension to the Taxonomy

The new semantic features not supported by the taxonomy of the ADM can all be classified as types of restrictions on data types. Thus, the starting point proposed here for extending the taxonomy of the ADM to support the new semantics changes the taxonomy mentioned above from a triple to a quad consisting of { **Degree, Heterogeneity, Construction, Restriction** }. **Restriction** would be a list of all active types of restrictions on a domain. Possible values include:

- *Value*, which indicates restrictions on the possible values of the domain
- *Degree*, which indicates a restriction on the specific number of elements
- *Derivation*, which indicates that the type cannot be used as a supertype for another type, or there are restrictions on how a base type of the domain may modify inherited attributes
- *None*, which indicates no restrictions

Only one of each of these **Restriction** values may be present in the list. For example, if an integer domain is restricted to values less than 10 and greater than 0, then the *Value* **Restriction** indicates both bounds.

The *Value* **Restriction** is a restriction of the possible values of a domain. *Value* may be used if the **Heterogeneity** is *Atomic*. If the **Heterogeneity** were *Structure*, then *Value* would not make sense, because each domain listed in the *Structure* might treat the *Value* constraint differently. Likewise, *Value* is not appropriate if **Heterogeneity** is *Union*. To restrict the values in a structure or union type, the domains that are included would need to have the *Value* **Restriction**.

The *Degree* **Restriction** is a restriction of the number of elements of a domain. *Degree* may be used if the **Degree** is *Multiple*. If the **Degree** is *Single*, then number of elements is exactly one, so specifying a *Degree* **Restriction** would be redundant.

The *Derivation* **Restriction** is a restriction on how a domain may be used as a supertype in inheritance. *Derivation* either indicates that the domain may not be used a supertype for another subtype, or that any of its subtypes are restricted in how they may modify certain inherited attributes. A Java class that has the 'Final' keyword is:

{ *Single/Multiple, Atomic/Structure, Recursive,* [*Derivation*] }

The *None* **Restriction** indicates no restriction on the composition or use of the domain. The *None* **Restriction** is mutually exclusive with all other **Restriction** values. The *None* **Restriction** is used when representing a data structure that could be denoted in the original ADM such as the C data structures mentioned above:

Union	{ *Single, Union, Recursive,* [*None*] }
Structure	{ *Single, Structure, Recursive,* [*None*] }
Array	{ *Multiple, Atomic, Recursive,* [*None*] }

Here the original triple that was used in Sanderson [19] makes up the first part of the triple, and the value [None] indicates there are no restrictions.

A domain may have multiple restrictions, which would be indicated by more than one value for **Restriction**. For example, I could define a single-digit number type that derives from integer and could not act as a base class for another class. Such a type would be categorized in the new taxonomy as follows:

{ *Single, Atomic, Recursive,* [*Value, Derivation*] }

5.3 – New Focus for the ADM

The taxonomy of the original ADM contains an implicit assumption that virtually any data type can be used in constructing a new data type if the new data type is Recursive. With the addition of **Restriction**, this may not be the case. Some domains may have restrictions that affect how that domain is used (as is the case with *Derivation*) or constructed (as is the case with *Degree* or *Value*) which prevent or restrict reuse. This necessitates a change in the construction

of new domains in the ADM. When a language such as Java uses a keyword ‘final’, then the taxonomy of the original ADM is no longer adequate to capture the semantics being used. By adding **Restriction** to the taxonomy, it becomes possible to track these semantic features. The new semantic features restrict how existing domains can be used to construct new domains, or how multiple items from a domain may be aggregated. Since the taxonomy of the ADM is being changed, the constructor, grouping, migrator, and inheritance functions would need to be updated to recognize **Restriction** and implement the new focus. The semantics of the data structures and schemas considered by Sanderson [19] are still fully supported by the new taxonomy of the ADM, since we have a *None* value for **Restriction**. In such cases, the ADM and the migrators should function exactly as they did in the original model.

Finally, all meaningful quads of the new taxonomy will be listed. The values for **Restriction** are constrained by the rules mentioned above.

{ *Single*, *Atomic*, N/A , [*None*] }
{ *Single*, *Atomic*, N/A , [*Value*] }
{ *Single*, *Atomic*, N/A , [*Derivation*] }
{ *Single*, *Atomic*, N/A , [*Value*, *Derivation*] }
{ *Single*, *Union*, *Simple*, [*None*] }
{ *Single*, *Union*, *Simple*, [*Derivation*] }
{ *Single*, *Union*, *Recursive*, [*None*] }
{ *Single*, *Union*, *Recursive*, [*Derivation*] }
{ *Single*, *Structure*, *Simple*, [*None*] }
{ *Single*, *Structure*, *Simple*, [*Derivation*] }
{ *Single*, *Structure*, *Recursive*, [*None*] }
{ *Single*, *Structure*, *Recursive*, [*Derivation*] }
{ *Multiple*, *Atomic*, N/A , [*None*] }
{ *Multiple*, *Atomic*, N/A , [*Value*] }
{ *Multiple*, *Atomic*, N/A , [*Degree*] }
{ *Multiple*, *Atomic*, N/A , [*Derivation*] }
{ *Multiple*, *Atomic*, N/A , [*Value*, *Degree*] }
{ *Multiple*, *Atomic*, N/A , [*Value*, *Derivation*] }
{ *Multiple*, *Atomic*, N/A , [*Degree*, *Derivation*] }
{ *Multiple*, *Atomic*, N/A , [*Value*, *Degree*, *Derivation*] }
{ *Multiple*, *Union*, *Simple*, [*None*] }
{ *Multiple*, *Union*, *Simple*, [*Degree*] }
{ *Multiple*, *Union*, *Simple*, [*Derivation*] }
{ *Multiple*, *Union*, *Simple*, [*Degree*, *Derivation*] }
{ *Multiple*, *Union*, *Recursive*, [*None*] }
{ *Multiple*, *Union*, *Recursive*, [*Degree*] }

{ *Multiple*, *Union*, *Recursive*, [*Derivation*] }
{ *Multiple*, *Union*, *Recursive*, [*Degree*, *Derivation*] }
{ *Multiple*, *Structure*, *Recursive*, [*None*] }
{ *Multiple*, *Structure*, *Recursive*, [*Degree*] }
{ *Multiple*, *Structure*, *Recursive*, [*Derivation*] }
{ *Multiple*, *Structure*, *Recursive*, [*Degree*, *Derivation*] }

CHAPTER 6

IMPLEMENTATION

6.1 – ADM Changes

The ADM, as presented by Sanderson [19], consists of an eight-tuple: the state S, the initial state P, the manipulators M, the oid space O, the constructors C, grouping operators G, the alphabet A, and the inheritance operators I. The extended ADM proposed in Chapter 5 adds a **Restriction** qualifier that places additional constraints on the instantiation, use, and content of a model's domains (i.e., data types). Three sets of revisions would be needed to incorporate this Restriction qualifier into an implementation of Sanderson's ADM: one involving the implementation's manipulators, a second involving the implementation's inheritance operators, and a third involving the implementation's constructor and grouping operators.

The set of manipulators M would need to be modified to recognize and enforce constraints that *Value* and *Degree* **Restrictions** can place on a domain's values. *Value* and *Degree* constraints provide for the substitutability of an instance of a subtype for an instance of its supertype. For example, if a type B has the *Degree* **Restriction** because it is derived from type A by adding a restriction on the number of elements, then instances of type B are substitutable for instances of type A. Then the manipulators for B that are derived from the manipulators for A must contain rules that enforce this restriction.

The inheritance operators, I, would have to be modified to recognize and enforce the constraints that the *Derivation* **Restriction** can place on a domain's values. For example, when building an inheritance hierarchy under the new model, the inheritance operators should check for the presence of the *Derivation* **Restriction** in a prospective base domain before adding the new domain to the state S. Also, they must recognize the presence of *Value* and *Degree* restrictions. If *Value* and *Degree* are not present, then the inheritance relationship being used is assumed to be a traditional inheritance relationship where the subtype extends the supertype.

Finally, the constructors C and the grouping operators G also have to be modified to recognize **Restriction**. For the constructors, a domain may have a *Value* **Restriction** that must

be recognized and enforced in the implementation of the constructor. The grouping operators need to be implemented to recognize the *Degree* **Restriction**, since it inherently deals with the number of items in a collection data structure.

The other four elements of the ADM would not change significantly in implementing support for **Restriction**. The concept of the state S and initial state P would not change, although the ADM would have to be able to denote **Restriction** for domains.

The notion of the alphabet A also has not changed. Although some languages like Java support Unicode variable names, instead of just ACII, this has no bearing on the types of semantic features being considered. Furthermore, the alphabet still serves the same purpose, providing the ability to create a unique name for each domain.

The oids space O consists of unique object identifiers that are used internally in the ADM to track the definition of data types. This functionality is unaffected by the changes that have been proposed to the ADM.

6.2 – ASM and Migrators

The operators mentioned above deal with representing the semantic features of a schema. In order to actually predict semantic information loss in translation between programming and specification languages, import and export functions to a common data model must be defined along with migration operators on that common model, and the semantic effects of the changes that occur when those migration operators are applied needs to be tracked. In Sanderson [19], an Abstract Semantic Model tracked the effects of migrator functions performing a translation. Although the migrators themselves and the ASM are outside the scope of this work, it is useful to consider what difficulties might be encountered in implementing the migrators, and what semantic loss can be expected in translations involving the languages considered above. The expected difficulties in translation and necessary semantic loss are detailed below in a series of tables, with each table looking at one quad from the taxonomy and predicting how translation of a domain characterized by that quad to each of the eight languages might proceed.

Table 1- Expected loss in translation of a 'Final' class.

	‘**Final’ Class:** { *Single/Multiple, Atomic/Structure, Recursive,* [*Derivation*] }
Python	Python does not support the *Derivation* **Restriction**. A translation of a domain of this type to Python will result in { *Single/Multiple, Atomic/Structure, Recursive,* [*None*] }
Java	Java supports this type of domain.
PHP	PHP does not support the *Derivation* **Restriction**. A translation of a domain of this type to PHP will result in { *Single/Multiple, Atomic/Structure, Recursive,* [*None*] }
C# .NET	C# .NET supports this type of domain.
VB.NET	VB.NET supports this type of domain.
Oracle OO Extensions	Oracle OO Extensions do not support the *Derivation* **Restriction**. A translation of a domain of this type would map most closely to ‘Type’, which is { *Single, Atomic/Structure, Recursive,* [*None*] }, although ‘VARRAY’ or ‘Nested Table’ is possible, both of which are { *Multiple, Atomic/Structure, Recursive,* [*None*] }
Prolog	Prolog does not support the *Derivation* **Restriction**. It also has data structures that map only very roughly onto classes. Translation to Prolog would be most closely accomplished by ‘Fact’ which is { *Single, Atomic/Structure/Union, Recursive,* [*None*] }. If the domain is multiple (a collection class), then it is only possible to come close if the collection class is *Atomic* with Prolog’s ‘List’, which is { *Multiple, Atomic, Simple/Recursive,* [*None*] }. If, however, the collection class is *Structure,* then the domain is lost almost entirely, because ‘List’ of Prolog is not indexed, so no technique like correlated arrays could be used.
XML Schemas	XML Schemas support this type of domain.

Table 2- Expected loss in translation of a *Degree* restricted subtype.

	'Degree' restricted subtype, substitutable for its supertype. { *Multiple, Atomic/Structure, Recursive,* [*Degree*] }
Python	Python supports structures that are { *Multiple, Atomic/Structure, Recursive,* [*None*] }. There is no static support for the *Degree* **Restriction** in Python. Runtime checks (such as Assert statements) could be used to enforce such semantics.
Java	Java supports structures that are { *Multiple, Atomic/Structure, Recursive,* [*None*] }. There is no static support for the *Degree* **Restriction** in Java. Although arrays in Java are declared with a specific degree, arrays do not support such restrictions as part of the domain. Runtime checks (as part of the methods of a collection class) could be used to enforce such semantics.
PHP	PHP supports structures that are { *Multiple, Atomic/Structure, Recursive,* [*None*] }. There is no static support for the *Degree* **Restriction** in PHP. Runtime checks could be used to enforce such semantics.
C# .NET	C# .NET supports structures that are { *Multiple, Atomic/Structure, Recursive,* [*None*] }. There is no static support for the *Degree* **Restriction** in C# .NET. Runtime checks (as part of the methods of a collection class) could be used to enforce such semantics.
VB.NET	VB.NET supports structures that are { *Multiple, Atomic/Structure, Recursive,* [*None*] }. There is no static support for the *Degree* **Restriction** in VB.NET. Runtime checks (as part of the methods of a collection class) could be used to enforce such semantics.
Oracle OO Extensions	Oracle OO Extensions supports structures that are { *Multiple, Atomic/Structure, Recursive,* [*None*] }. There is no static support for the *Degree* **Restriction** in Oracle OO Extensions. Triggers or constraints could be used to enforce such semantics.
Prolog	Prolog supports structures that are { *Multiple, Atomic, Recursive,* [*None*] }. There is no static support for the *Degree* **Restriction** in Prolog. If the collection is *Structure,* then the domain is lost almost entirely, because 'List' of Prolog is not indexed, so no technique like correlated arrays could be used.
XML Schemas	XML Schemas support this type of domain.

Table 3- Expected loss in tranlsation of a *Value* restricted subtype, with 'Final' keyword.

	"SingleDigitInteger" type with 'Final' keyword { *Single, Atomic, Recursive,* [*Value, Derivation*] }
Python	Python supports neither *Value* nor *Derivation* **Restriction**. Python supports structures that are { *Single, Atomic, Recursive,* [*None*] }. Runtime checks (such as Assert statements) could be used to enforce the *Value* **Restriction**.
Java	Java supports the *Derivation* **Restriction**, but not the *Value* **Restriction**. Java supports structures that are { *Single, Atomic, Recursive,* [*Derivation*] }. Runtime checks could be used (as part of the methods of a "SingleDigitInteger" class) to enforce the *Value* **Restriction**.
PHP	PHP supports neither *Value* nor *Derivation* **Restriction**. PHP supports structures that are { *Single, Atomic, Recursive,* [*None*] }. Runtime checks could be used to enforce the *Value* **Restriction**.
C# .NET	C# .NET supports the *Derivation* **Restriction**, but not the *Value* **Restriction**. C# .NET supports structures that are { *Single, Atomic, Recursive,* [*Derivation*] }. Runtime checks (as part of the methods of a "SingleDigitInteger" class) could be used to enforce the *Value* **Restriction**.
VB.NET	VB.NET supports the *Derivation* **Restriction**, but not the *Value* **Restriction**. VB.NET supports structures that are { *Single, Atomic, Recursive,* [*Derivation*] }. Runtime checks (as part of the methods of a "SingleDigitInteger" class) could be used to enforce the *Value* **Restriction**.
Oracle OO Extensions	Oracle OO Extensions supports structures ('Type') that are { Single, *Atomic, Recursive,* [*None*] }. There is no static support for the *Value* **Restriction** in Oracle OO Extensions. Triggers or constraints could be used to enforce such semantics in a database using Oracle OO Extensions. There is no support for the *Derivation* **Restriction**.
Prolog	Prolog supports structures that are { *Single, Atomic, Recursive,* [*None*] }. Prolog does not support inheritance, so the *Value* and *Derivation* **Restrictions** have no meaning in Prolog.
XML Schemas	XML Schemas supports this type of domain.

CHAPTER 7

CONCLUSIONS

Semantic modeling of programming and specification languages is a process of determining what features of languages are interesting or useful and trying to express those features in a model. In the present work, the Abstract Data Model, used in Sanderson [19] for predicting semantic loss in translation between computer languages, was used to categorize eight languages that Sanderson did not consider. Four of the languages were found to support semantic features that could not be expressed in the old model, so modifications were suggested and the implementations were discussed.

The new semantic features discovered in these languages can be classified into two types of restrictions on inheritance relationships. The first type of restriction prevents a class from being used as a base class for another class, as is the case in Java when the 'final' keyword is used. The second, more fundamental type of restriction restricts supertype-subtype relationships. In the traditional inheritance seen in languages like C++ a subtype is always an extension (data or functionality) of a supertype. In XML schemas, subtyping can restrict the values allowable in instances of a type (Super type Number, Subtype Integer for example); this ensures that instances of the subtype are substitutable for instances of the supertype.

Domains were classified in the original ADM by their position in a taxonomy, which consisted of a triple of { **Degree**, **Heterogeneity**, **Construction }**. The ADM was extended to support the semantic restriction features by adding a fourth item to the taxonomy, making a quad of { **Degree**, **Heterogeneity**, **Construction**, **Restriction }**, where **Restriction** is a list of all active semantic features that restrict inheritance for a domain. The possible values for **Restriction** are *None,* indicating no restrictions, *Derivation,* indicating 'final' keyword type restrictions on derivation from the domain, *Value*, indicating that the domain derives from its supertype by restricting possible values, and *Degree,* indicating that the domain derives from its supertype by restricting the degree.

Actual implementations of the changes were beyond the scope of this work as they involved the Abstract Data Model, the Abstract Semantic Model, and migrator functions that operate on the schemas. The current work dealt only with the identification of new semantic features appropriate to the ADM and suggested changes to support those features. The newly identified features are appropriate for inclusion in an extended ADM, because these features characterize inheritance between domains, an issue that was explicitly considered by Sanderson [19].

There are several avenues for further research based on this work. The first, and most obvious, is to implement the ASM and ADM and migrators of Sanderson [19] along with the modifications to the ADM suggested in the present work. Second, programming languages will continue to evolve, and there may even already be some that support semantic features that are appropriate to add to the ADM. A third way of continuing the research would to be to re-examine the languages already reviewed and refine or adapt the ADM to track different types of information that might be lost in translation between computer languages. As an abstract model, the ADM focuses on certain features of languages while ignoring others. The decisions made about what types of features the ADM should handle are not the only ones that might be of interest. Finally, the ADM could be made extensible to allow it to denote semantic features not found in current languages, rather than just those features found in the languages that have been considered.

REFERENCES

[1] Abiteboul, S. and Hull, R. IFO: A Formal Semantic Database Model. ACM Transactions on Database Systems. Volume 12, Issue 4, December 1987, Pages 525-565.

[2] Appleman, Dan. Moving to VB.NET: Strategies, Concepts, and Code. Apress. 2001. 560 pages.

[3] Bradley, Neil. The XML Schema companion. Pearson Education, Inc. 2004. 312 pages.

[4] Codd, E. F. Extending the database relational model to capture more meaning. ACM Transactions on Database Systems. Volume 4, Issue 4, December 1979, Pages 397-434.

[5] C# Programmer's Reference. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vcoriCProgrammersReference.asp Retrieved April 2005.

[6] Date, C. J. An Introduction to Database Systems: Seventh Edition. Addison Wesley Longman, Inc. 2000. 938 pages.

[7] Horstmann, Cay. Computing Concepts With Java 2 Essentials, Second Edition. John Wiley & Sons, Inc., 2000. 762 pages.

[8] Hull, R. and Yap, C. K. The Format Model: A Theory of Database Organization. JACM. Volume 31, Issue 3, July 1984, Pages 518-544.

[9] Language Syntaxes of VB.NET and C# - DotNet Zone - DNzone.COM. http://www.dnzone.com/ShowDetail.asp?NewsId=356 Retrieved April 2005.

[10] McLeod, D. and Smith, J. M., Abstraction in Databases. Proceedings of the 1980 workshop on Data abstraction, databases and conceptual modeling. 1980, Pages 19-25.

[11] Oid (Java 2 Platform SE v1.4.2). http://java.sun.com/j2se/1.4.2/docs/api/org/ietf/jgss/Oid.html Retrieved April 2005.

[12] Oracle9i Application Developer's Guide - Object-Relational Features. http://www.cs.umb.edu/cs634/ora9idocs/appdev.920/a96594/toc.htm Retrieved April 2005

[13] PHP – Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/PHP_programming_language Retrieved April 2005.

[14] PHP: PHP Manual – Manual. http://www.php.net/manual/en/index.php Retrieved April 2005.

[15] Pressman, Roger S. Software Engineering: A Practitioner's Approach, Fifth Edition. McGraw-Hill. 2001. 860 pages.

[16] Python Documentation Index. http://www.python.org/doc/ Retrieved April 2005.

[17] Rob, Peter and Coronel, Carlos. Database Systems: Design, Implementation, & Management Sixth Edition. Thomson Course Technology. 2004. 824 pages.

[18] Saint-Dizier, Patrick. An Introduction to Programming in Prolog. Springer-Verlag New York, Inc. 1990. 184 pages.

[19] Sanderson, D. Loss of Data Semantics in Syntax Directed Translation. Ph.D. Dissertation. Rensselaer Polytechnic Institute. September, 1994. 421 pages.

[20] Smith, J., and Smith, D. Database Abstractions: Aggregation and Generalization. ACM Transactions on Database Systems, Volume 2, Issue 2, June 1977, Pages 105-133.

[21] Su, Stanley Y. W., et. al. OSAM*.KBMS: an object-oriented knowledge base management system for supporting advanced applications. Proceedings of the 1993 ACM SIGMOD international conference on Management of data. 1993, Pages 540-541.

[22] The XML Schema Tutorial. http://www.w3schools.com/schema/default.asp Retrieved April 2005.

[23] Troelsen, Andrew. C# and the .NET Platform. Apress. 2001. 970 pages.

VITA

Matthew Bryston Winegar

Education:	Public Schools, Kingsport, Tennessee
	Emory University, Atlanta, Georgia
	Physics, B.A., 2001
	East Tennessee State University, Johnson City, Tennessee
	Computer Science, M.S., 2005
Professional Experience:	Undergraduate Researcher, University of Florida, Summer 1999
	Graduate Assistant, East Tennessee State University, Department of Computer and Information Sciences, 2003-2005