Reusing (Shrink Wrap) Schemas by Modifying Concept Schemas

Lois Delcambre and Jimmy Langston

Technical Report No. CS/E 95-009

June 1995

Reusing (Shrink Wrap) Schemas by Modifying Concept Schemas

Lois Delcambre
lmd@cse.ogi.edu

Jimmy Langston
jtl@cse.ogi.edu

Data Intensive Systems Center
Dept. of Computer Science and Engineering
Oregon Graduate Institute
Portland, OR 97291

June 12, 1995

Abstract

We provide mechanisms that facilitate database design based on a shrink wrap schema. A *shrink wrap schema* is a well-crafted, complete, global schema that represents an application. We develop the notion of concept schemas as a way to decompose shrink wrap schemas. A *concept schema* is a subset of an application schema that addresses one particular point of view in an application. To aid in shrink wrap schema-based design, we define schema modification operations to customize each concept schema, to match the designer's perception of the application. We maintain the integrated, customized user schema. We enforce consistency checks to provide feedback to the designer about interactions among the concept schemas. We embody these mechanisms in an interactive system that aids in shrink wrap schema-based design.

Our approach simplifies database design, particularly for a complex data model and/or complex schema. Our main contribution lies in the ability to use a shrink wrap schema as a starting point for modeling an application area. The shrink wrap schema approach promotes reuse of past design efforts; prior approaches to schema reuse do not attempt to reuse an entire schema nor do they focus on local customization. The focus of this paper is on the definition of concept schemas and their corresponding modification operations.

Keywords: schema reuse, schema modification, database design

1 Introduction

Higher level constructs better represent application concepts by making explicit the structure and semantics of the concepts. However, those same constructs can be more difficult to understand and use correctly. They add to the complexity of the data model, require more design effort, and raise issues regarding interaction among constructs. Another problem is that a global schema, by its very nature, integrates all views or perspectives. This implies that global schemas can be difficult to understand and to modify.

1.1 The Problem

We define a *shrink wrap schema* [1] to be a well-crafted, complete, global schema that represents an application. Ideally, we could reduce database design to a trivial process by providing a "standard" schema for each application area. A standard schema facilitates data interchange and provides an accurate representation of semantics. Using a shrink wrap schema may be easier than designing an application schema from scratch because the standard schema correctly uses the data model constructs and provides the designer with examples relevant to the application.

Realistically, we know that it is difficult to provide a standard schema for an application area because particular applications need different variations of the schema. The scope of the system under construction may differ from that of the standard schema. The business details or other aspects of an application might be simpler or more elaborate than the standard schema. This need for variations of the standard schema might defeat the use of shrink wrap schemas. For these reasons, the use of a shrink wrap schema for database design dictates that provision be made for modifying the schema to allow representations to vary.

Another problem is that when considering a shrink wrap schema (or any global schema), the designer is likely to be overwhelmed when given the entire schema at once. It may be difficult to make changes directly to the global schema because of the number and complexity of the constructs. Therefore, it is useful for the designer to be able to consider the shrink wrap schema a piece at a time to avoid these difficulties. We define concept schema types in order to systematically decompose the shrink wrap schema for viewing and modification.

1.2 The Solution

We develop the notion of concept schemas as a way to decompose shrink wrap schemas. A *concept schema* is a subset of an application schema that addresses one particular point of view, i.e., one concept, in an application. The concept schemas correspond to modeling abstractions provided by the data model (e.g., aggregation, generalization). To aid in shrink wrap schema-based design, we define techniques to customize each concept schema through simplification and elaboration, to match the designer's perception of the application. This process alleviates some of the rigidity forced on a designer when trying to use a shrink wrap schema in an application. After the schema has been tailored to the designer's needs, we construct the global, customized user schema. We also provide consistency checks based on the concept schema structure and semantics, in order to discover problems in the user schema and to provide feedback to the designer about interactions among the various concept schemas.

This research project includes the definition and development of an interactive schema designer with a schema repository to support shrink wrap schema-based design using concept schemas [29]. The focus of this paper is on:

- the definition of generic structure patterns (types) for concept schemas,

[1] This term originated in conversations with Michael Brodie of GTE Laboratories.

- the definition of the elaboration and simplification operations for concept schemas (i.e., the definition of schema modification operations).

2 Related Work

This work is inspired by the large number of complex semantic and object-oriented data models that are available today for the database designer [12, 22, 27, 24, 39, 35, 36, 31, 28, 26, 21, 34, 7, 14, 30]. While the increased complexity of the models produce more accurate representations of the application, the data models are more difficult to utilize. Tools for database design have been proposed to aid the designer, but each of them starts the design process from scratch [33, 38, 37, 9] Metamodels are defined by researchers to classify and describe data models or schemas at a higher level of abstraction [1, 23, 2]. Our work is similar to this research only in that the concept schema types are defined at the metamodel level.

Our work is closely related to research on reusing schemas. Work in this area concentrates on using portions of schemas as modeling constructs. Most approaches proceed by extracting portions of schemas from several similar schemas and describing the common information represented at a higher level of abstraction. This results in a generic pattern, abstracted from one or more applications, that is hopefully useful during the design of new schemas. Our approach is centered around the reuse of a shrink wrap schema for an application area. We believe that a well-crafted schema within an application is likely to be (re)used. The concept schemas are defined based on generic structure, but are always a subset of an application schema.

Peter Coad suggests using patterns as the building blocks for object-oriented design and construction [15]. He states that a pattern standardizes small piecework into a larger chunk or unit. Patterns are discovered among low-level elements by looking at the relationships between them. In object-oriented technology, he identifies patterns of relationships as generalization, specialization, whole-part, association, and messaging. In addition, he feels that there exist other patterns that are applicable multiple times in a single application and across different kinds of applications. He defines an object-oriented pattern as a small grouping of classes that is likely to be helpful again and again in object-oriented development. Coad presents seven different patterns along with guidelines for when they are useful [15]. Patterns as the units of reuse in this context involve multiple classes, relationships, and message passing. Our concept schema types are defined around what he calls patterns of relationships. He identifies some random patterns that occur in schemas whereas our concept schema types are based on the data model constructs and cover the entire schema.

Antonellis et al. [10] refer to information system design moving towards the design by reuse paradigm. In this approach, applications are developed not from scratch, but by tailoring and personalizing reusable components. This work proposes a methodology for building a library of entity-relationship schemas and extracting reusable components from selected schemas. A reusable component is a generic entity with an associated meta-entity that provides guidelines for reuse in an application. The schemas are organized into a library with each schema characterized by a schema descriptor, based on structural characteristics of the schema. Schemas of the same application are clustered based on their schema

descriptors. A metric called semantic affinity is proposed for describing the similarity between two entities within the clusters. If the pairs of entities are sufficiently close to each other, they are presented to the application engineer for possible construction of a generic entity. The generic entity is defined by factoring out the common attributes of the pair of entities [10]. Patterns in this context are based on entities as the unit of reuse. Reuse is supported in our approach occurring at the concept schema level as well as at the shrink wrap schema level.

Bertram [4] defines a schema-pattern as a subset of an abstract data model that can be instantiated for use in application-specific models. He specifies the components necessary to specify a schema pattern including: a data model for pattern specification, a specification of the pattern boundary and interface across the boundary, rules for pattern usage, a description of some pattern variants, and links to similar patterns. In addition, the behavior of the pattern may be specified with basic functions and a state model [4]. This information is stored in a pattern library. However, no examples of schema patterns are presented in this paper. The main contribution of the paper is quality criteria for the evaluation of schema patterns, similar to those for reuse of software components. In comparison to this work, concept schemas are not schema patterns. Concept schemas are based on schema structure, but they are used as a formal mechanism for decomposing a shrink wrap schema.

3 Shrink wrap schema customization with concept schemas

The focus of the research is on the schema customization process based on a shrink wrap, or predefined, standard schema for an application. We therefore assume that a well-designed, richly-structured, standard schema has been created before the beginning of the design process. A first step in the process requires a decomposed or concept schema-based representation of the shrink wrap schema.

During shrink wrap schema-based design, the designer considers the concept schemas one by one to determine how they relate to the application being designed. If the concept is deemed useful, the concept schema can be customized for the designer's application. Designers are allowed to customize all of the concept schema types and may elaborate and simplify the information specified in the interface definitions. There is a set of modifications available to the designer and each change to the concept schema is subject to constraints defined over the operation applied. For example, some operations possible on an interface definition include adding or removing object types, relationships, and attributes.

The basic function of the schema customization process developed in this research is to allow the user, i.e. the schema designer, to view and alter the standard, well-crafted or shrink wrapped schema. The main components of the system are shown in Figure 1. The schema repository is a knowledge base for the entire process. It holds the original shrink wrap schema used as the starting point, the concept schemas (generated from the shrink wrap schema), the workspace for the schema under design, the custom schema, and the mapping from the original to the custom schema. The schema repository also includes a knowledge component that specifies consistency rules, propagation rules, and constraints used to generate feedback for the designer.

The designer interacts directly with the interactive schema designer by issuing operations (to view and alter the schema) and by receiving feedback. The feedback consists of error or informational messages

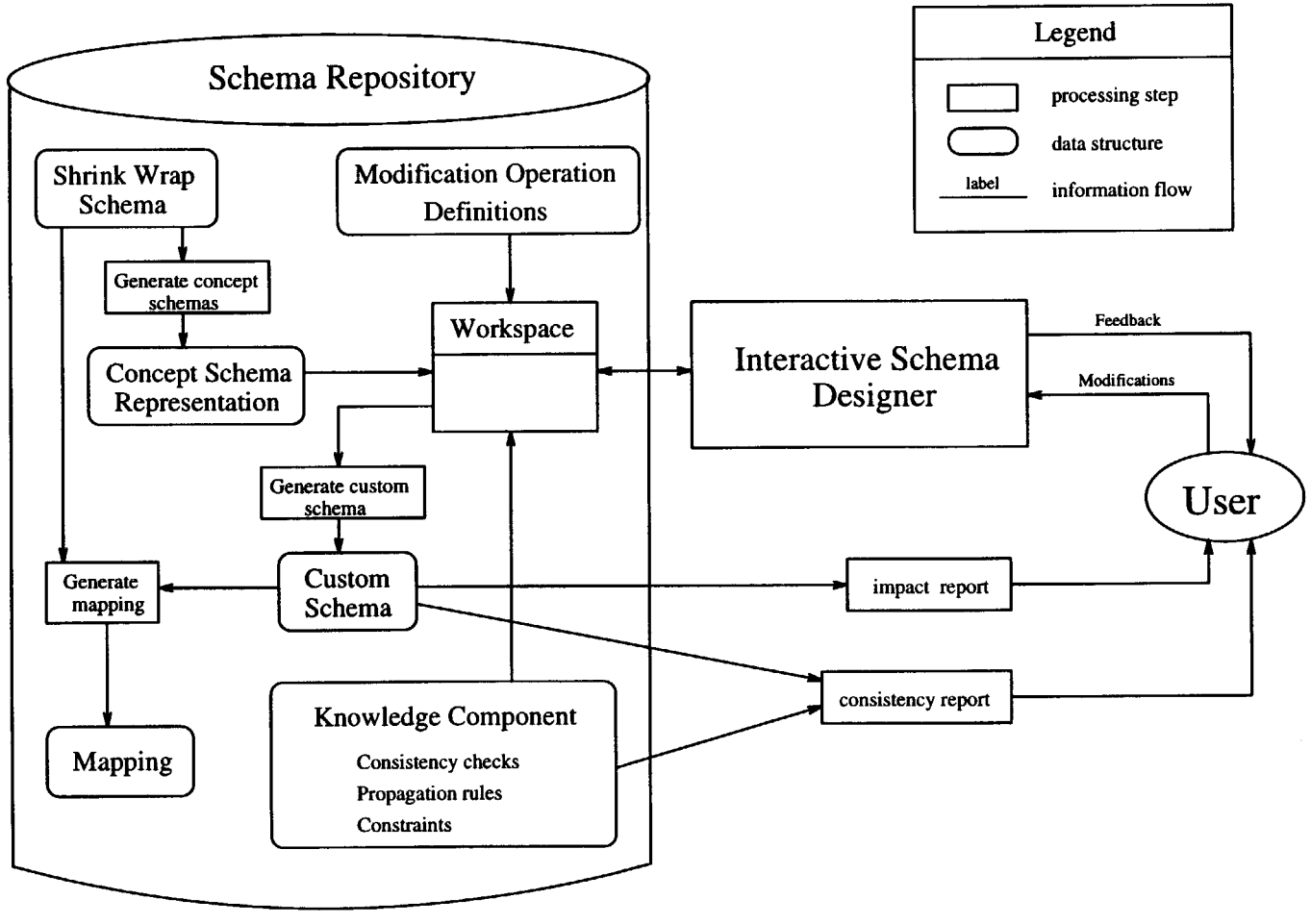

Figure 1: Shrink wrap schema customization

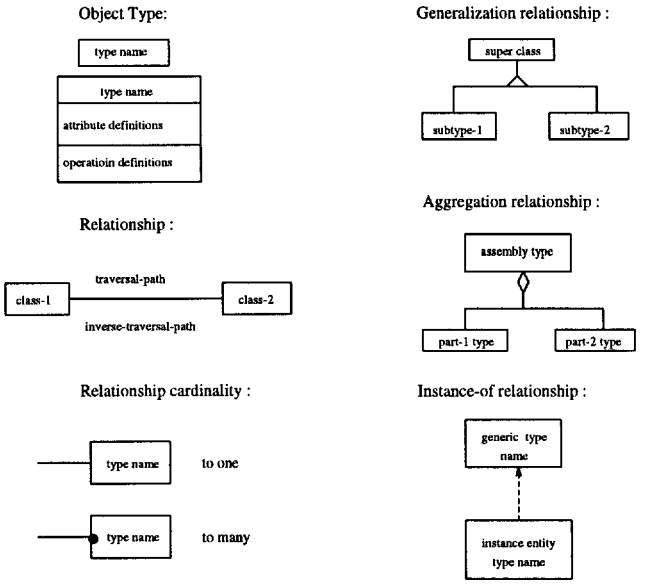

Figure 2: Object Modeling Technique notation

about the requested operations. The schema designer provides a graphical interface for modifying the concept schemas to match the user's application. The schema designer uses windows and pull-down menus to guide the designer through modifications. The possible modifications are restricted according to the concept schema type that is being modified. Constraints are enforced on the user's changes to ensure that the schema specification is syntactically and semantically correct.

3.1 Notation

We adopt Rumbaugh's notation for the graphical specification of the structure of our schemas because it has a rich, structural model. The structural notation is summarized on the inside of the front cover of [34] and is partially reproduced in Figure 2.

The ODMG standards [11] provide an Object Model, an Object Definition Language (ODL) to define interfaces to object types that conform to the Object Model, as well as C++ and Smalltalk bindings. We adopt ODL as the data definition language on which our concept schemas are formally defined and the schema modifications are discussed. We choose ODL because it is language and database system independent. The grammar is extended slightly in this work to support the instance-of and aggregation relationship types, currently not present in the Object Model or the ODL. The extended ODMG data model includes the relationships of supertype, aggregation, and instance-of.

The part-of relationship is used to express a relationship between an entity and its components. This relationship is prevalent in CAD applications where physical products and their components are being designed and represented. The relationship has an implicit 1:N cardinality between the whole and its components.

The instance-of relationship is used to express a relationship between a generic specification entity and specific instances of that entity. This relationship also has an implicit 1:N cardinality. The relationship

is between one generic specification and many instances of that specification.

3.2 Simplifying Assumptions, Justifications

In this section, we present the assumptions made in the research.

- Good faith use - By deleting the entire shrink wrap schema and adding a completely new schema with some of the same concepts expressed or named differently, a designer can lose many of the benefits that our approach provides. We assume that he/she doesn't.
- Name equivalence - We assume name equivalence of object types, attributes, relationships, and operations. This is a simplifying assumption, but we believe that our work could be easily adjusted to remove this assumption.
- Uniqueness - We assume that the object type names, relationship names, and attribute names uniquely identify the type definitions, relationships, and attributes, respectively. The operation names are unique as well, except in the case where an operation is overridden.
- Semantic stability - Attributes, relationships, and methods are moved only within the generalization hierarchy established by the shrink wrap schema. We believe that information that is moved around in the schema should move between object types that are semantically similar. This implies that the moves should be within the generalization hierarchy. For example, a legal move might be to move an attribute up the hierarchy to reside in a supertype's interface definition.
- Entity Stability - An entity in the standard schema is an entity in the custom schema (or it has been deleted). This assumption dismisses from consideration, for example, operations that would change an entity's representation in the shrink wrap schema to an attribute in the custom schema.
- Relationship, Attribute, Method Stability - Like entity stability, each of these constructs are either present or absent in the custom schema, but not represented in some other structural manner. However, they could be moved within the limits of semantic stability explained above.
- Single root for generalization hierarchies - Each generalization hierarchy contains a single root by which it can be uniquely identified. Any hierarchy with two or more roots can be easily transformed by creating an abstract supertype of the multiple roots.

3.3 Concept schema types

This section defines the concept schema types defined in this research. They are based on the modeling abstractions present in the extended ODL. It is possible to algorithmically decompose a schema defined in extended ODL into concept schemas.

3.3.1 Wagon wheel concept schema types

The basic building block for schemas is the wagon wheel concept schema type. The wagon wheel concept schema type is inspired by object type definitions in semantic and object-oriented data models. A wagon

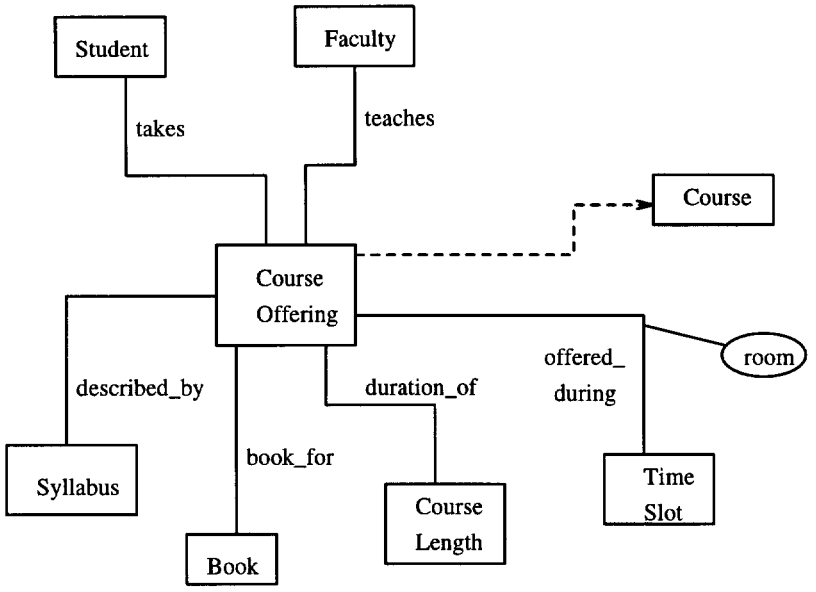

Figure 3: Course Offering Concept Schema.

wheel concept schema consists of one object type that serves as the focal point of the wagon wheel and supporting attributes and relationships that emanate from the focal point. We believe that each focal point of a wagon wheel corresponds to a different point of view in the schema, centered on one object type.

At least one wagon wheel concept schema exists for every object type in a shrink wrap schema. It is possible for different points of view of an object type to result in more than one concept schema having the same focal point. The union of all the initial concept schemas gives the original shrink wrap schema. Structurally, the wagon wheel concept schema type, in addition to the focal point, includes objects that are just one relationship away from the focal point. A wagon wheel may center on object types that participate in generalization, aggregation, or instance-of hierarchies. Generalization, aggregation, and instance-of links of distance one from the focal point can be included in a wagon wheel because the wagon wheel schema includes allowable relationship types. An example of a wagon wheel concept schema that represents a course offering in a university setting is given in Figure 3. The course offering object type is the focal point of the concept schema. The relationships and attributes for the course offering form the "spokes" of the wagon wheel. The figure gives a high level view of a course offering and does not relay all the information expressible in the data model (e.g., cardinalities, attributes, operations). The dotted line represents an instance-of relationship between a course and an offering of the course.

Work by Aranow on class-centered modeling [25] provides design techniques centered around classes. He believes that classes represent concepts and that schemas should be presented to users one class at a time with the class containing the supporting information similar to our description. His research provides for reuse of design efforts with the class as the unit of reuse.

Although we believe that wagon wheels (i.e., classes) are the basic building block for object-oriented schemas, we are also interested in additional structure and semantics that occur beyond wagon wheels.

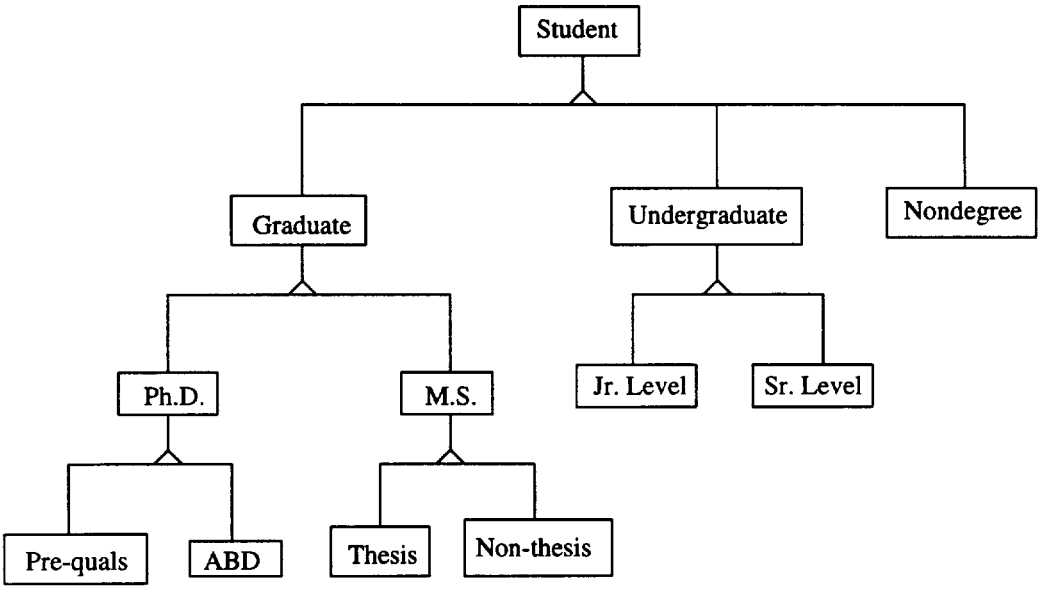

Figure 4: Student Generalization Hierarchy

We have identified three additional concept schema types in the chosen object model.

3.3.2 Generalization hierarchies

A generalization hierarchy specifies the object types that participate in subtype/supertype relationships and inherit information from supertype to subtype. Defining a concept schema type for generalization hierarchies (where other attributes and relationships are not shown) gives the designer an integrated view of the inheritance paths between object types. For example, consider the generalization hierarchy of students in the university conceptual schema given in Figure 4. The student generalization hierarchy shows, for example, that a Non-thesis masters student object inherits the attributes and operations defined on a Graduate student object type.
Each generalization concept schema describes all subclasses of the root type and allow the schema designer to consider the inheritance patterns, distinctly from the various wagon wheels.

3.3.3 Aggregation hierarchies

The aggregation hierarchy expresses part-of relationships between two object types. The part-of relationship has a 1:N cardinality by definition. This relationship is mentioned as a possible extension to the Object Model [11] and is present in some semantic data models. Figure 5 is an example of a parts explosions for a lumber yard. The construction supplies necessary to build a house, for instance, can be recorded with the roof of the house consisting of plywood decking, tar paper, and shingles. These types of hierarchies are useful in VLSI and CAD applications.
We propose a rooted aggregation hierarchy as one of our generic concept schema patterns. This concept schema allows the designer to consider the part-of explosion for each aggregated object and provide a point of view that is different from wagon wheels and different from the generalization patterns.

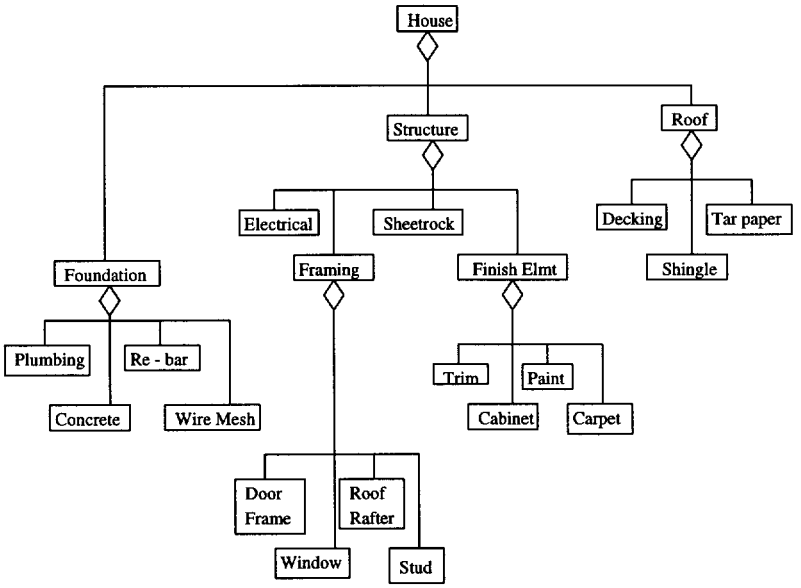

Figure 5: House Aggregation Hierarchy

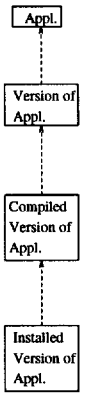

Figure 6: Software Instance-of Sequence

3.3.4 Instance-of Hierarchy

There is a benefit to viewing a sequence of several instance-of relationships between object types as a concept schema. The software version application concept schema from the Environmental and Molecular Sciences Laboratory (EMSL) [40, 32, 17, 16, 19] schema is shown in Figure 6. This example can be best understood by presenting objects for each of the object types. For example, the *C compiler* is an application object that is related to many versions of C compilers including *version 3.0.* The *version 3.0* may have been compiled on many different machines, each compilation creating a *compiled version 3.0* executable (a compiled version of the application object). The executable is in turn installed on many machines, each installation creating a *installed version 3.0* (an installed version of application object). We propose a generic concept schema pattern to describe the sequence of instance-of links. In our experience, the instance-of hierarchy has been linear with no branches. However, we are not claiming that a branched structure is not possible.

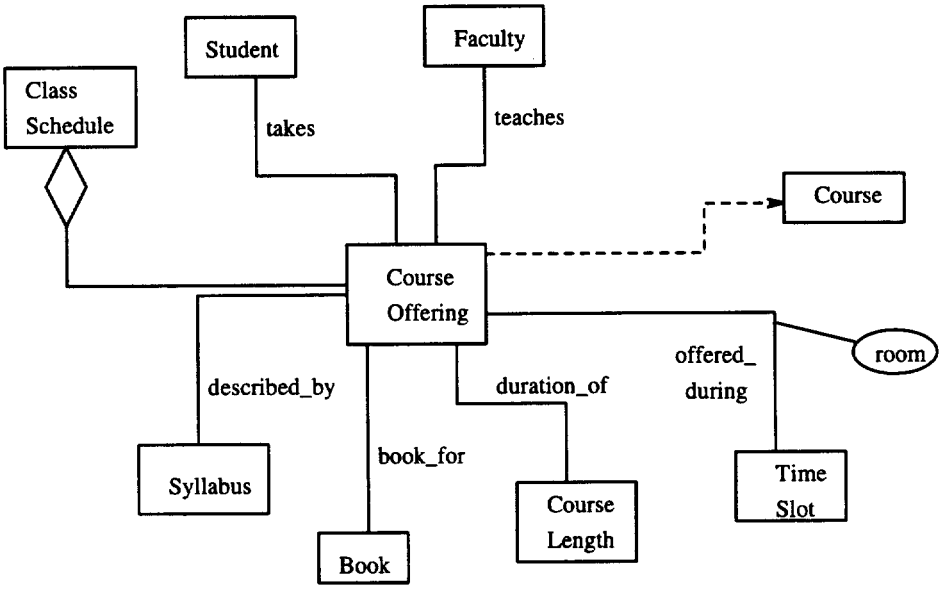

Figure 7: Elaborated Course Offering

3.4 Schema modifications

We propose to formalize the simplification/elaboration process for concept schemas by restricting the operations that can be performed. The BNF grammar in Appendix A specifies the syntax for a complete set of schema modification operations. We initiate the discussion by considering an example that illustrates a schema modification.

Consider the course offering concept schema from Figure 3. The designer may want to include a course schedule object type that consists of course offerings. In that case, the course offering concept schema would need to be elaborated by including an aggregation link from class schedule to course offering. The resulting course offering concept schema is shown in Figure 7. Consider another situation where courses are offered by correspondence only. In this case, the course offering concept schema is simplified by removing the time slot entity and room attribute.

We make some restrictions on the modifications in the context of shrink wrap schemas that simplify our problem from that of schema evolution:

- We do not allow modifications that change the name of interfaces, relationships, and operations. This decision reflects our name equivalence assumption.
- We only allow information in modify operations to move within the generalization hierarchies. This restricts the movement of information among similar object types, providing what we term *semantic stability.*

Table 1 summarizes the operations that are allowed in the concept schemas for each of the ODL candidates for modification. The letters A, D, and M are abbreviations for Add, Delete, and Modify. The instance-of hierarchies allow the designer to modify information having to do with the instance-of

relationships. The designer is allowed to add and delete object types and connect them with instance-of relationships or modify the information (excluding names) about the existing relationships. The designer can also add and delete instance-of relationships between object types.

	Wagon Whl	Gen. hier.	Part-of	Instance-of
Interface Definition	A/D	A/D	A/D	A/D
Type name				
Type Properties				
Supertype (ISA)	A/D	A/D/M		
Extent name	A/D/M			
Key list	A/D/M			
Instance Properties				
Attribute	A/D	M		
Type	M			
Size	M			
Name				
Relationship	A/D			
Target type		M		
Traversal path name				
Inverse path name				
One way cardinality	M			
Order by list	M			
Operation	A/D	M		
Return type	M			
Name				
Argument list	M			
Exceptions Raised	M			
Part-of relationship	A/D		A/D	
Target type			M	
Traversal path name				
Inverse path name				
One way cardinality			M	
Order by list			M	
Instance-of Relationship	A/D			A/D
Target type				M
Traversal path name				
Inverse path name				
One way cardinality				M
Order by list				M

Table 1: Operations on ODL schema definitions in the context of concept schema types. Note: disallowed operations support name equivalence

The part-of hierarchies allow modification of the existing part-of relationships as well as operations to re-wire the hierarchy. These operations include the ability to add and delete object types and part-of relationships.

The generalization hierarchies provide several interesting operations to move information around within the schema. Operations, attributes, and relationships are allowed to change object types within the

hierarchy. Each of these operations causes the inheritance pattern of information to change within the hierarchy. Object types can be added and deleted in the hierarchy and supertype relationships can be added, deleted, and modified for re-wiring the generalization hierarchy.

The largest portion of the modifications are supported in wagon wheel concept schemas. The wagon wheel concept schemas are the best choice for most modifications because they cover the entire shrink wrap schema and they are centered around the definition of an object. Object types can be added and deleted and the complete set of operations for the type properties, extent name and key list, are allowed. With respect to instance properties; attributes, relationships, and operations can be added and deleted and existing instance information can be modified. However, modification of supertypes, part-of relationships, and instance-of relationships is not supported in wagon wheel concept schemas; those modifications must be addressed in their respective concept schema types.

As an example, consider the modify_relationship_target_type operation. This operation allows the designer to move the end of the relationship that is defined in an object type that participates in a generalization hierarchy up or down the generalization hierarchy. Consider the situation in Figure 8 where a department has an employee and the employee works in a department. Now suppose that students also work in departments, so modify the target type of works_in_a from employee to person to represent this information. The relationship for Department and Employee interface definition was originally specified in the interface definition as:

```
Department:
   relationship set<Employee> has inverse
   Employee::works_in_a
Employee:
   relationship Department works_in_a inverse
   Department::has
```

and, after the operation modify relationship target type (Employee, works_in_a, Person), now becomes :

```
Department:
   relationship set<Person> has inverse
   Person::works_in_a
Person:
   relationship Department works_in_a inverse
   Department::has
```

We exclude operations that split and merge interface definitions. We believe that it is more appropriate to subtype the interface definitions to be split and to create an abstract supertype for interface definitions to be merged.

3.5 Completeness of this approach

We consider whether schema modification using our approach allows a designer to make all possible modifications (i.e., that he/she would be able to make if he/she were modifying a schema without the

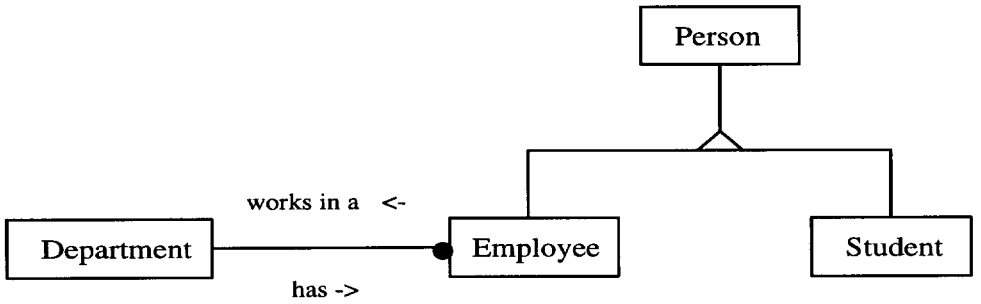

Figure 8: Modify target type example

benefit of our approach). Based on the syntax of ODL, we have enumerated every possible construct that can be modified in an ODL specification.

We consider first whether all constructs expressible in ODL are available for add, delete, and modify operations within our approach. Consider the information in Table 2 for addition of the ODL constructs; the deletion operations are identical, with the word "add" changed to "delete" in the operation name. These tables show our operations that cover the addition and deletion of the ODL candidates for modification.

Now we consider more complicated changes to the schema. Do we prevent the user from the changing the schema in some way? Based on our assumptions, we intentionally limit the modification operations. For example, it is not possible to replace an entity in the shrink wrap schema by an attribute value in the customized schema (and retain the semantic connection between the two). However, we observe that any construct present in the shrink wrap schema can be deleted and any new construct can be added. In the extreme case, the entire shrink wrap schema can be deleted, and an entirely new (custom) schema can be added. This defeats any automatic semantic integration between the two schemas. On the other hand, it demonstrates that our approach does not prevent the user from creating any possible schema. The modify operations are summarized in Table 3. This table does not have to be complete, because, as stated above, all schema modifications can be made using only add and delete operations. Note that names are not allowed to be modified in accordance with our assumptions of uniqueness and equivalence of names.

4 Real-world example of shrink wrap schema based design

ACEDB provides an example of a system that has been adapted for reuse in many different applications. This system and its successors furnish a scenario where shrink wrap schema-based design has been performed manually, with a number of benefits, without using our research. We demonstrate here that the kind of changes made to ACEDB are admissible using our technology.

ACEDB is an application, with an internal database, originally developed to study the physical mapping data for the nematode genome project. The application was reused for several related projects, resulting in a family of related, customized schemas based on the original schema. [5, 8, 13, 20, 18]. Designers who were building similar databases picked up the ACEDB system and modified it for their purposes. The fact that the systems must all use the same display code requires the schemas to be similar.

	Operation
Interface Definition	Add_type_definition
Type name	Add_type_definition
Type Properties	
Supertype (ISA)	Add_supertype
Extent name	Add_extent_name
Key list	Add_key_list
Instance Properties	
Attribute	Add_attribute
Type	Add_attribute
Size	Add_attribute
Name	Add_attribute
Relationship	Add_Relationship
Target type	Add_Relationship
Traversal path name	Add_Relationship
Inverse path name	Add_Relationship
One way cardinality	Add_Relationship
Order by list	Add_Relationship
Operation	Add_operation
Return type	Add_operation
Name	Add_operation
Argument list	Add_operation
Exceptions Raised	Add_operation
Part-of relationship	Add_part-of_relationship
Target type	Add_part-of_relationship
Traversal path name	Add_part-of_relationship
Inverse path name	Add_part-of_relationship
One way cardinality	Add_part-of_relationship
Order by list	Add_part-of_relationship
Instance-of Relationship	Add_instance-of_relationship
Target type	Add_instance-of_relationship
Traversal path name	Add_instance-of_relationship
Inverse path name	Add_instance-of_relationship
One way cardinality	Add_instance-of_relationship
Order by list	Add_instance-of_relationship

Table 2: Addition operations on ODL candidates

	Operation
Interface Definition	
Type name	
Type Properties	
Supertype (ISA)	Modify_supertype
Extent name	Modify_extent_name
Key list	Modify_key_list
Instance Properties	
Attribute	Modify_attribute
Type	Modify_attribute_type
Size	Modify_attribute_size
Name	
Relationship	
Target type	Modify_Relationship
Traversal path name	
Inverse path name	
One way cardinality	Modify_Relationship_cardinality
Order by list	Modify_Relationship_order_by
Operation	Modify_operation
Return type	Modify_operation_return_type
Name	
Argument list	Modify_operation_arg_list
Exceptions Raised	Modify_operation_exceptions_raised
Part-of relationship	
Target type	Modify_part-of_target_type
Traversal path name	
Inverse path name	
One way cardinality	Modify_part-of_cardinality
Order by list	Modify_part-of_order_by
Instance-of Relationship	
Target type	Modify_instance-of_target_type
Traversal path name	
Inverse path name	
One way cardinality	Modify_instance-of_cardinality
Order by list	Modify_instance-of_order_by

Table 3: Modify operations on ODL candidates

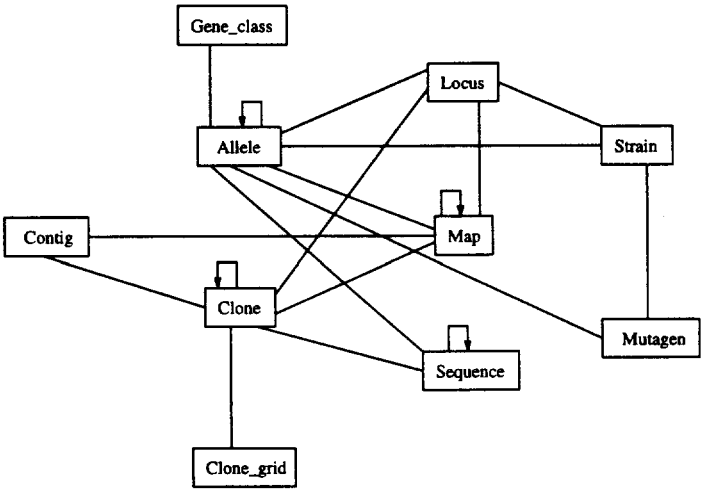

Figure 9: ACEDB object types

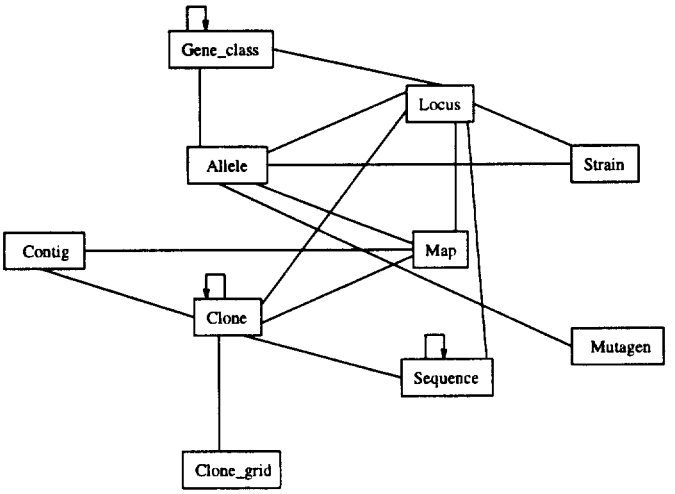

Figure 10: SacchDB object types

We have studied the schemas from two systems based on ACEDB; the Arabidopsis database (AAtDB) for the plant thale cress, and the Saccharomyces database (SacchDB) for yeast. We have examined the common classes in the three schemas to determine the similarity of the system schemas. Figures 9, 10, and 11 give a subset of the object types that have the same name across the schemas along with their interconnections by relationships. The object types have the same name and further study of the type definitions reveals that much of the structure is the same. The object type *strain* in the ACEDB schema and *phenotype* in the AAtDB schema are semantically equivalent terms that are used in the animal and plant disciplines.

This scenario gives empirical evidence that shrink wrap schema-based design is feasible. A shrink wrap schema based on the ACEDB schema could have been constructed and each of the later physical mapping databases could have used our mechanisms to create the custom schema for their application.

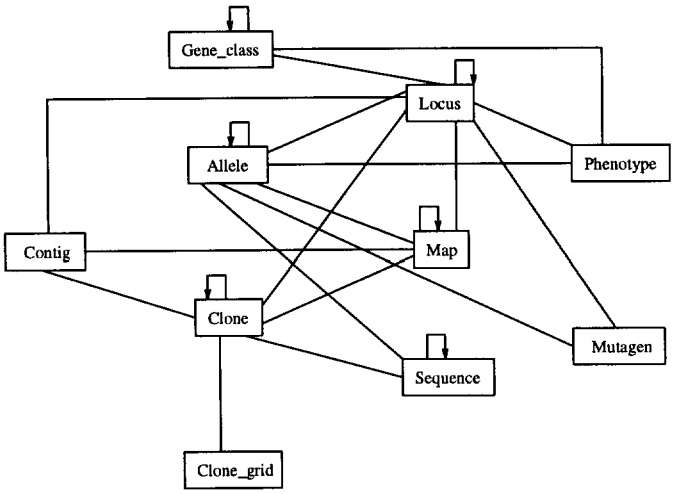

Figure 11: AAtDB object types

5 Evaluation and Conclusions

First, we consider limitations to our work. Name equivalence means that things with the same name are the same and things with different names are different. We acknowledge that database designers are very likely to want to introduce local names for constructs that appear in the schema. The extension of our work to handle this possibility requires that the user indicate a change of name, and that the system maintain the mapping from shrink wrap schema names to local names. We view such an extension as straightforward and not mainstream to our work.

We assume that the object type names, relationship names, and attribute names uniquely identify the type definitions, relationships, and attributes, respectively. This assumption complements name equivalence in helping establish equivalence among schema information. We do not deal with establishing equivalence of information that does not have the same name. Some models require that attribute names, for example, be unique across the entire schema. Others require only that they be unique within objects. In the latter case, an attribute name must be qualified by the object type name where the attribute resides. Once again, we believe that the extension of our work to allow qualified names is straightforward. Note that Rebecca Wirfs-Brock et al. recommend the use of unique names during design to improve the clarity of the object model [42].

We restrict the movement of attributes, relationships, and methods to be within the generalization hierarchy established by the shrink wrap schema. We make this choice on semantic (not simplifying) grounds. We would argue, for example, that replacing a participant in a relationship with an object type that is not semantically comparable results in a semantically distinct relationship.

An entity in the shrink wrap schema is either an entity in the custom schema or not in the custom schema at all. This assumption is in line with our belief that the shrink wrap schema is a well-crafted representation of the application. More sophisticated modifications would allow, for example, an object type in the shrink wrap schema to be replaced by an attribute value or by the union of several object types. We believe that entity, relationship, attribute, and method stability contributes to the designer's

ability to understand the shrink wrap schema and its relationship to the custom schema. Thus, although it is a simplifying assumption, we believe that the changes we allow are quite intuitive. We also see generalization of our current approach to a more expressive set of mappings as a possibility for future work.

Second, we consider the contribution of this work. Consider the example of building a latex document for the first time. The task is easier if you are allowed to start with a latex file and modify it for your application. In fact, a sample latex file that demonstrates basic functionality is provided with the tutorial for using latex. The same principle holds in database design. People comprehend examples and the shrink wrap schema provides those examples in a form relevant to their application. A contribution of our research lies in the ability to use a shrink wrap schema as a starting point for modeling an application area. The shrink wrap schema provides a well-crafted schema for the designer and ultimately leads to better schema quality for the custom schema. Other contributing factors to the quality of the custom schema include the guidance and feedback provided by the interactive schema designer; this aspect of the work is discussed elsewhere [29]. Schema quality of the shrink wrap schema can be improved by revising the representation over time as it is employed and reviewed by diverse design teams.

Software reusability, like object-orientation, impacts everything from management practices to software development standards [3]. Software practitioners often limit their definition of software to source and object code, but, in truth, software includes many other things such as modules, documentation, plans, standards, analysis and design products, and quality assurance efforts. Software reusability can have a very positive impact on software reliability, efficiency, and time to market [3]. Our research provides reuse of the conceptual schema for database systems. The design of the successors of ACEDB [5, 8, 13, 20, 18] that have been produced manually, could, in fact, have been created using our technology.

Since our research operates using the implementation independent ODL, the techniques discussed are broadly applicable. Our approach is not dependent on a DBMS or even a data model. The data model provides an object-oriented approach, but there has been work, for example, on modeling in an object-oriented model and translating the results to other models such as entity relationship diagrams and relational models.

Our approach simplifies the customization process for the designer by restricting the modification operations available, according to the situation. The modifications are also organized and checked by the interactive schema design system to provide guidance and feedback to the designer. The language that is created for specifying modifications formalizes the modification choices for implementation in a system. These factors accelerate the process of schema design.

Possible applications of our work are: to facilitate interoperation through common objects. Work in progress [6] is attempting to establish a Business Object Model to promote the conduct of business over the network. In general, systems built from the same shrink wrap schema (i.e., common objects) can be integrated for information interchange because the semantically identical constructs have already been identified.

This paper reports on research on concept schemas and their associated modification operations that are part of a larger research project. The complete list of activities is shown here:

1. Modifications to ODL to accommodate part-of and instance-of relationships.

2. Definition of generic structure patterns (types) for concept schemas.

3. Algorithms for extracting concept schemas from an ODL specification.

4. Architecture of an interactive tool for shrink wrap schema-based design.

5. Restriction of the schema modification operations possible on an ODL specification for shrink wrap schema customization.

6. Definition of schema modification operations in the context of concept schemas.

7. Definition of the language for specifying the modification operations.

8. Discussion of the semantics and definition of constraints associated with the operations.

9. Definition of a set of constraints enforced in the interactive tool for shrink wrap schema-based design and classification of the constraints into logical categories.

    - Definition of a set of rules to show the designer the impact of the proposed modification operation (i.e., all of the changes that follow from a given change).
    - Definition of a set of cautionary statements to the user in the form of feedback

10. Definition of a mapping representation that records the semantic correspondence between the shrink wrap and customized schema.

11. Specification of an approach to generating deliverables for designer feedback as a result of shrink wrap schema customization.

12. Development of a prototype implementation to demonstrate the feasibility of the research.

    - Implementation of the schema repository as an Object Store application.
    - Construction of a parser for Object Store header files.
    - Implementation of mapping generation, user deliverables generation, constraint enforcement.

    The prototype is operational, but we have not fully implemented the interactive nature of the schema modification operations.

This paper focuses on 1,2,5,6, and 7.
Future work will include a full-featured implementation of the interactive schema designer with interactive feedback, impact (perhaps shown in color on the graphical view), and cautionary statements, coupled with a field trial of the technology. Possible extensions to this work include:

- An explanation facility for the existing concept schemas can be created to explain the information represented in the concept schema to the designer.

- More complex schema modification operations with well-defined semantics can be added to provide guidance to the designer. The modifications can be incorporated into the schema designer along with expected constraints and impact on the schema.

- Additional constraints and rules for the knowledge component can be added to aid in the design process. These would make life easier for the designer.
- Constraint Analysis [41] can be used in the consistency check to suggest the operations that need to be altered to enforce semantic constraints.
- It may be worthwhile to include the object oriented type constructors (set-of,list-of,bag-of,array-of), for construction of complex objects, in the data model. If that is the case, then the object-oriented type constructors should be represented in concept schemas; we feel that they may be implemented as a variation of aggregation.

References

[1] Carlo Batini, Giuseppe Di Battista, and Giuseppe Santucci. Structuring primitives for a dictionary of entity relationship data schemas. *IEEE Transactions on Software Engineering*, 19(4):344–365, April 1993.

[2] Z. Bellahsene. An active meta-model for knowledge evolution in an object-oriented database. In Collette Rolland, Francois Bodart, and Corine Cauvet, editors, *Proceedings of the Fifth International Conference, CAiSE '93*, pages 39–53. Springer-Verlag, June 1993.

[3] Edward V. Berard. *Essays on Object-oriented Software Engineering*, volume 1. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.

[4] Martin Bertram. Data modelling in the large. *SIGMOD Record*, 23(4):8–12, December 1994.

[5] Martin J. Bishop, editor. *Guide to Human Genome Computing.* Academic Press, San Diego, CA, 1994.

[6] OMG BOMSIG. Proposal Draft, December 1994.

[7] G. Booch. *Object-oriented Design with Applications.* Benjamim Cummings, 1991.

[8] David Boone. Oregon Graduate Institute, 1995. Correspondence.

[9] M. A. Casanova, A. L. Furtado, and L. Tucherman. A software tool for modular database design. *ACM Transactions on Database Systems*, 16(2):209–234, June 1991.

[10] S. Castano, V. De Antonellis, and B. Zonta. Classifying and reusing conceptual schemas. In G. Pernul and A. M. Tjoa, editors, *ER '92 – Entity Relationship Approach*, pages 121–138. Springer-Verlag, October 1992.

[11] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, San Mateo, CA, 1.1 edition, 1994.

[12] P. P. Chen. The entity-relationship model – towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[13] Mike Cherry. Stanford University, 1995. AAtDB and SacchDB schemas.

[14] P. Coad and E. Yourdon. *Object-oriented Analysis.* Yourdon Press, 1991.

[15] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–159, September 1992.

[16] J. B. Cushing, D. Maier, M. Rao, D. M. DeVaney, and D. Feller. Object-oriented database support for computational chemistry. In *Sixth International Working Conference on Statistical and Scientific Database Management (SSDBM)*, June 1992.

[17] Judith Bayard Cushing, David Hansen, David Maier, and Calton Pu. Connecting scientific programs and data using object databases. *IEEE Bulletin of the Technical Committee on Data Engineering*, 16(1):9–13, March 1993.

[18] Richard Durbin and Jean Thierry-Mieg. http://moulon.inra.fr/acedb/acedb.html. World Wide Web, 1994.

[19] J.C. French, editor. *Computational Proxies: Modeling Scientific Applications in Object Databases.* Seventh International Working Conference on Statistical and Scientific Database Management (SSDBM), IEEE, September 27-29 1994.

[20] Nat Goodman. Massachusetts Institute of Technology, 1995. Correspondence.

[21] K. Gorman and J. Choobineh. An overview of the object-oriented entity-relationship model (oOERM). In *Proceedings of the Twenty-third Annual Hawaii International Conference on System Sciences*, 1991.

[22] Michael Hammer and Dennis McLeod. Database description with sdm: A semantic database model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[23] Shuguang Hong and Fred Maryanski. Using a meta model to represent object-oriented data models. In Z. Michalewicz, editor, *Proceedings of the IEEE Data Engineering Conference (DEC)*, pages 11–19, 1990.

[24] Richard Hull and Roger King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[25] Andrew T. F. Hutt, editor. *Object Analysis and Design: Description of Methods.* John Wiley and Sons, New York, 1994.

[26] G. Kappel and M. Schrefl. A behavior integrated entity-relationship approach for the design of object-oriented databases. In C. Batini, editor, *Proceedings of the Seventh International Conference on Entity-Relationship Approach*, pages 175–192, 1988.

[27] L. Kerschberg and J. E. S. Pacheco. A functional database model. Technical report, Pontifica Universidade Catolicalo Rio de Janeiro, Brazil, February 1976.

[28] C. S. Ku, C. Youn, and H. J. Kim. An object-oriented entity-relationship model. In *1991 ISMM International Conference on Computer Applications in Design, Simulation, and Analysis*, pages 55–58, March 1991.

[29] Jimmy Langston. *Object-Oriented Database Design based on a Shrink Wrap Schema.* PhD thesis, University of Southwestern Louisiana, 1995.

[30] J. Martin and J. Odell. *Object-oriented Analysis and Design.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1992.

[31] S. B. Navathe and Pillalamarri. OOER: Toward making the e-R approach object- oriented. In *Proceedings of the Eighth International Conference on Entity-Relationship Approach*, pages 55–76, 1989.

[32] Pacific Northwest Laboratory, P.O. Box 999, Richland, WA 99352. *Environmental and Molecular Sciences Laboratory.*

[33] Arnon Rosenthal and David Reiner. Tools and transformations – rigorous and otherwise – for practical database design. *ACM Transactions on Database Systems*, 19(2):167–211, June 1994.

[34] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented Modeling and Design.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

[35] Peretz Shoval. Essential information structure diagrams and database schema design. *Information Systems*, 10(4):417–423, 1985.

[36] Peretz Shoval. An integrated methodology for functional analysis, process design and database design. *Information Systems*, 16(1):49–64, 1991.

[37] Peretz Shoval, Ehud Gudes, and Moshe Goldstein. GISD: A graphical interactive system for conceptual database design. *Information Systems*, 13(1):81–95, 1988.

[38] Il-Yeol Song and E. K. Park. Object-oriented database design methodologies. In Timothy W. Finin, Charles K. Nicholas, and Yelena Yesha, editors, *Proceedings of the First International CIKM*, pages 115–140. Springer-Verlag, 1992.

[39] T. J. Teorey, D. Yang, and J. P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18(12):197–222, June 1986.

[40] United States Department of Energy, P.O. Box 550, Richland, WA 99352. *Overview of the 1993 Hanford Site-Specific Plan*, March 1993.

[41] Susan D. Urban and Lois M. L. Delcambre. Constraint analysis: Identifying design alternatives for operations on complex objects. *Transactions on Knowledge and Data Engineering*, 2(4):391–400, December 1990.

[42] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-oriented Software.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.

A BNF grammar for schema modifications

This grammar specifies the syntax of the operations allowed on the concept schemas. The inputs of the operations are "hooks" into the ODL grammar specification; therefore, this grammar is built on top of the ODL specification grammar. This language formalizes the specification of the schema modification operations for an interactive tool for shrink wrap schema based design. Some of the information that is necessary for the modification specification is implicit in the grammar. For example, the concept schema type that the operation is defined over is specified through nonterminals in the grammar. This information would be known in the interactive schema designer since the user would be working with that concept schema type when the operation is issued. Also, all of the inputs to the operation do not have to be explicitly entered by the designer since some may be inferred from the context in which the operation is issued.

```
<concept_schema_type> ::= <wagon_wheel> | <generalization_hierarchy>
        | <aggregation_hierarchy> | <instance-of_hierarchy>
<wagon_wheel> ::= <type_definition_ops>
                  | <ww_type_property_ops> | <ww_instance_property_ops>
<generalization_hierarchy> ::= <type_definition_ops>
                               | <gh_type_property_ops>
                               | <gh_instance_property_ops>
<aggregation_hierarchy> ::= <type_definition_ops>
                            | <ah_instance_property_ops>
<instance-of_hierarchy> ::= <type_definition_ops>
                            | <ih_instance_property_ops>

<type_definition_ops> ::= <add_type_definition>
                          | <delete_type_definition>

<ww_type_property_ops> ::= <ww_supertype_ops> | <ww_extent_ops>
                           | <ww_key_list_ops>
<gh_type_property_ops> ::= <gh_supertype_ops>
```

```
<ww_supertype_ops> ::= <add_supertype> | <delete_supertype>
<ww_extent_ops> ::= <add_extent_name> | <delete_extent_name>
                  | <modify_extent_name>
<ww_key_list_ops> ::= <add_key_list> | <delete_key_list>
                    | <modify_key_list>
<gh_supertype_ops> ::= <add_supertype> | <delete_supertype>
                     <modify_supertype>

<ww_instance_property_ops> ::= <ww_attribute_ops>
                             | <ww_relationship_ops>
                             | <ww_operation_ops>
                             | <ww_part-of_ops>
                             | <ww_instance-of_ops>
<gh_instance_property_ops> ::= <gh_attribute_ops>
                             | <gh_relationship_ops>
                             | <gh_operation_ops>
<ah_instance_property_ops> ::= <ah_part-of_ops>
<ih_instance_property_ops> ::= <ih_instance-of_ops>

<ww_attribute_ops> ::= <add_attribute>
                     | <delete_attribute>
                     | <modify_attribute_type>
                     | <modify_attribute_size>
<ww_relationship_ops>::= <add_relationship>
                       | <delete_relationship>
                       | <modify_relationship_cardinality>
                       | <modify_relationship_order_by>
<ww_operation_ops> ::= <add_operation> | <delete_operation>
                     | <modify_operation_return_type>
                     | <modify_operation_arg_list>
                     | <modify_operation_exceptions_raised>
<ww_part-of_ops> ::= <add_part-of_relationship>
                   | <delete_part-of_relationship>
<ww_instance-of_ops> ::= <add_instance-of_relationship>
                       | <delete_instance-of_relationship>
<gh_attribute_ops> ::= <modify_attribute>
<gh_relationship_ops> ::= <modify_relationship_target_type>
<gh_operation_ops> ::= <modify_operation>
<ah_part-of_ops> ::= <add_part-of_relationship>
                   | <delete_part-of_relationship>
                   | <modify_part-of_target_type>
                   | <modify_part-of_cardinality>
                   | <modify_part-of_order_by>
<ih_instance_of_ops> ::= <add_instance-of_relationship>
                   | <delete_instance-of_relationship>
                   | <modify_instance-of_target_type>
                   | <modify_instance-of_cardinality>
```

```
                    | <modify_instance-of_order_by>

/* operation specifications */
<add_type_definition> ::= add_type_definition ( <type_name> )
<delete_type_definition> ::= delete_type_definition ( <type_name> )
<add_supertype> ::= add_supertype ( <type_name>
                    <supertype> )
<delete_supertype> ::= delete_supertype ( <type_name>, <supertype> )
<modify_supertype> ::= modify_supertype ( <type_name>,
                      <old_supertype_list>, <new_supertype_list> ) /* re-wiring isa */
<old_supertype_list> ::= <supertype_list>
<new_supertype_list> ::= <supertype_list>
<add_extent_name> ::= add_extent_name ( <type_name>,
                      <extent_name> )
<delete_extent_name> ::= delete_extent_name ( <type_name>,
                      <extent_name> )
<modify_extent_name> ::= modify_extent_name ( <type_name>,
                         <old_extent_name>, <new_extent_name> )
<old_extent_name> ::= <extent_name>
<new_extent_name> ::= <extent_name>
<add_key_list> ::= add_key_list ( <type_name>,
                   <key_list> )
<delete_key_list> ::= delete_key_list ( <type_name>,
                      <key_list> )
<modify_key_list> ::= modify_key_list ( <type_name>,
                      <old_key_list>, <new_key_list> )
<add_attribute> ::= add_attribute ( <type_name>,
                    <domain_type>, [ <size> ], <attribute_name> )
<delete_attribute> ::= delete_attribute ( <type_name>,
       <attribute_name> )
<modify_attribute> ::= modify_attribute ( <type_name>,
       <type_name> ) /* move attr.  up/down gen.  hier */
<modify_attribute_type> ::= modify_attribute_type (
       <type_name>, <attribute_name>, <old_type>, <new_type> )
<old_type> ::= <domain_type>
<new_type> ::= <domain_type>
<modify_attribute_size> ::= modify_attribute_size (
       <type_name>, <attribute_name>, <old_size>, <new_size> )
<old_size> ::= <size>
<new_size> ::= <size>
<add_relationship> ::= add_relationship ( <type_name>,
    <target_of_path>, <traversal_path_name_1>
    <inverse_traversal_path>, [ <attribute_list> ] )
<delete_relationship> ::= delete_relationship (
      <type_name>, <traversal_path_name_1> )
<modify_relationship_target_type> ::= modify_relationship (
       <type_name>, <traversal_path_name_1>, <old_target_type>,
```

```
        <new_target_type> )
/* move relationship target up/down gen.  hier.  */
<old_target_type> ::= <target_type>
<new_target_type> ::= <target_type>
<modify_relationship_cardinality> ::=
        modify_relationship_cardinality ( <type_name>,
        <traversal_path_name_1>, <old_target_of_path>,
        <new_target_of_path> )
<old_target_of_path> ::= <target_of_path>
<new_target_of_path> ::= <target_of_path>
<modify_relationship_order_by> ::=
        modify_relationships_order_by ( <type_name>,
        <traversal_path_name_1>, <old_attr_list>,
<new_attr_list> )
<old_attr_list> ::= <attribute_list>
<new_attr_list> ::= <attribute_list>
<add_operation> ::= add_operation ( <type_name>, <return_type>,
     <operation_name>, [ <argument_list> ], [ <exceptions_raised> ] )
<delete_operation> ::= delete_operation ( <type_name>,
        <operation_name> )
<modify_operation> ::= modify_operation ( <type_name>,
        <operation_name>, <new_type_name> )
<new_type_name> ::= <type_name>
/* move operation up/down gen hier.  */
<modify_operation_return_type> ::= modify_operation_return_ type(
        <type_name>, <operation_name>, <old_return_type>,
        <new_return_type> )
<old_return_type> ::= <return_type>
<new_return_type> ::= <return_type>
<modify_operation_arg_list> ::= modify_operation_arg_list(
        <type_name>, <operation_name>, <old_arg_list>, <new_arg_list> )
<old_arg_list> ::= <argument_list>
<new_arg_list> ::= <argument_list>
<modify_operation_exceptions_raised> ::=
        modify_operation_exceptions _raised( <type_name>,
        <operation_name>, <old_exception_list>, <new_exception_list> )
<old_exception_list> ::= <exception_list>
<new_exception_list> ::= <exception_list>
<add_part-of_relationship> ::= <add_part-of_to_part_of_relationship>
     | <add_part-of_to_whole_relationship>
<add_part-of_to_part_of_relationship> ::=
     add_part-of_relationship ( <type_name>, <collection_type>,
     <target_type>, <traversal_path_name_1>,
     <inverse_traversal_path>, [ <attribute_list> ] )
<add_part-of_to_whole_relationship> ::  =
     add_part-of_relationship ( <type_name>, <target_type>,
     <traversal_path_name_1>, <inverse_traversal_path>,
```

```
    [ <attribute_list> ] )
<delete_part-of_relationship> ::= delete_part-of_relationship (
        <type_name>, <traversal_path_name_1> )
<modify_part-of_target_type> ::= modify_part-of_target_type (
        <type_name>, <traversal_path_name_1>, <old_target_type>,
        <new_target_type> )
<modify_part-of_cardinality> ::= modify_part-of_cardinality (
        <type_name>, <traversal_path_name_1>, <old_collect_type>,
        <new_collect_type> ) /* only allowed for to-part-of end */
<old_collect_type> ::= <collection_type>
<new_collect_type> ::= <collection_type>
<modify_part-of_order_by> ::= modify_part-of_order_by (
        <type_name>, <traversal_path_name_1>, <old_attr_list>,
        <new_attr_list> )
<add_instance-of_relationship> ::=
    <add_instance-of_to_instance_entities_relationship>
    | <add_instance-of_to_generic_entity_relationship>
<add_instance-of_to_instance_entities_relationship> ::=
    add_instance_of_relationship ( <type_name>,
    <collection_type>, <target_type>, <traversal_path_name_1>,
    <inverse_traversal_path>, [ <attribute_list> ] )
<add_instance-of_to_generic_entity_relationship> ::=
    add_instance-of_relationship ( <type_name>, <target_type>,
    <traversal_path_name_1>, <inverse_traversal_path>,
    [ <attribute_list> ] )
<delete_instance-of_relationship> ::=
        delete_instance-of_ relationship ( <type_name>,
        <traversal_path_name_1> )
<modify_instance-of_target_type> ::=
        modify_instance-of_target_type ( <type_name>,
        <traversal_path_name_1>, <old_target_type>,
        <new_target_type> )
<modify_instance-of_cardinality> ::=
        modify_instance-of_cardinality ( <type_name>,
        <traversal_path_name_1>, <old_collect_type>,
        <new_collect_type> )
        /* only allowed for to-instance-entities end of relationship */
<modify_instance-of_order_by> ::= modify_instance-of_ order_by (
        <type_name>, <traversal_path_name_1>, <old_attr_list>,
        <new_attr_list> )
```