CIFE CENTER FOR INTEGRATED FACILITY ENGINEERING

A Primitive-Composite Approach for Structural Data Modeling

H. Craig Howard, Jamal A. Abdalla and
D. H. Douglas Phan

TECHNICAL REPORT
Number 52

July 1991

Stanford University

If you would like to contact the authors please write to:

c/o CIFE, Civil Engineering,
Stanford University,
Terman Engineering Center
Mail Code: 4020
Stanford, CA 95305-4020

A Primitive-Composite Approach for Structural Data Modelling[1]

H. Craig Howard[2]   Jamal A. Abdalla[3]   D. H. Douglas Phan[4]

1 To appear in the *ASCE Journal for Computing in Civil Engineering*, Special Issue on Databases, 1991.

2 Assistant Professor, Department of Civil Engineering, Stanford University, Stanford, CA 94305-4020, USA; phone: (415) 723-5678; e-mail: hch@cive.stanford.edu.

3 Postdoctoral Research Affiliate, Department of Civil Engineering and Center for Integrated Facility Engineering, Stanford University, Stanford, CA 94305-4020, USA.

4 Graduate Research Associate, Department of Civil Engineering, Stanford University, Stanford, CA 94305-4020, USA.

Abstract

Accurate, efficient data exchange is vital to improving productivity in the architecture-engineering-construction industry. Much of the current work in data exchange is focused on the development of data exchange standards. In examining the data models in those standards and other efforts, we have identified a number of problem areas, including overuse of aggregation hierarchies, nonhomogeneous characterization hierarchies, and replacing relationships with subclasses. To support easy exchange of data between engineering computer applications while avoiding the data modelling pitfalls, we have developed the *primitive-composite* approach. The P-C approach provides a methodology for defining the primitive data concepts of a domain as separate objects and for assembling those concepts to form the composite abstractions that engineers customarily use. By relating composite objects from multiple programs to the primitive concepts, the data in those programs can be shared as primitive concepts derived from one program's composite object and reassembled as another program's composite object without having to provide a direct mapping between those two programs. The paper presents a formal definition of the primitive-composite approach and explores its application in structural engineering.

Contents

1. Introduction

The architecture-engineering-construction industry is still exchanging data as it did 100 years ago with paper drawings and reports. The introduction of the computer to the design process has changed the means of generating the paper, but it has not fundamentally changed the methods of sharing data across organizational boundaries. It is not uncommon to find that in one office a designer uses a powerful computer-aided design and drafting (CADD) package to produce a project drawing, while in another office, a construction estimator uses a digitizer to put the information from the same drawing back into a computer (in effect, *unCADD*). The result is a substantial net loss in efficiency and an increase in the ever-present potential for errors.

Therefore, we are faced with the problem of getting computer tools to communicate about engineering data. Whether the data is transferred directly or through a central database, dynamically or via batch files, the first problem is identifying the data to be exchanged in an integrated data model. A number of national and international efforts are underway to create integrated data models for the exchange of product data. However, the data modelling task in the architecture-engineering-construction domain is very difficult because so many different specialities are involved and because the life cycle is very long. Many different professionals, organizations, and programs contribute to and draw from the same overlapping set of data objects. In trying to satisfy all users of an engineering object, the natural tendency is to add more and more description to that object. The result is a set of extremely complex objects that no one specialist completely understands and a data model that is difficult (if not impossible) to understand, maintain, and extend.

To draw an analogy to human language, the result of the complex object approach is to produce a phrase book. As long as the phrase you need is in the book, you can communicate. However, if you need to communicate about something not in the book or even a subtle variation on a defined phrase, you are out of luck. Frequently a bilingual dictionary can be more useful by translating words rather than phrases. With the vocabulary of a language (along with the syntax), any phrase or sentence can be constructed, not just the limited set in the phrase book.

A data model composed of complex objects is like a phrase book—it contains a limited number of predigested ideas. With the primitive-composite approach described in this paper, we are defining a methodology for creating and applying a data "vocabulary" for a domain—a *primitive data model*—representing atomic concepts that can be arbitrarily combined. From this vocabulary, programmers and users can build the complex objects that they need and still communicate data by sharing the same primitive objects.

The purpose of this paper is to present the primitive-composite approach for data modelling and describe its application to structural engineering data. The following sections first review traditional data models (hierarchical, network, relational), object-oriented data models, and problem areas in engineering data modelling. Then the formal definition of the primitive-composite approach is presented, including descriptions of how to use it to build data models and exchange data. The last two sections discuss how the primitive-composite approach can be applied to structural engineering data and its overall merits for engineering data.

1.1 Data Models

A data model is a collection of conceptual tools for describing data in terms of data semantics, relationships, and constraints among data items (Tsichritzis and Lochovsky 1982). A data model thus defines rules according to which data *structures* are described, *constraints* among data items are imposed, and *operations* on data items are defined. Combinations of different rules regarding the definition of structures, constraints, and operations yield different data models.

The three dominant classical data models are the hierarchical, network, and relational models. The network data model organizes data records into an arbitrary graph of linked nodes (CODASYL 1971). Each node may contain more than one record. Each record in a network model is a collection of attributes of simple data types. Each link associates two related records. The major weakness of the network data model is that operations are constrained to follow strict paths defined by the node links, and that many-to-many relationships can not be represented directly. The hierarchical model is special case of the network model in which each node has only one parent (Date 1981). The hierarchical data model has severe limitations due to the fact that each node is restricted to one parent. The relational data model (Codd 1970) was introduced by E. F. Codd in 1970 and quickly gained popularity among business applications. In this model, similar objects are grouped into a single relation, which defines the attributes shared by all the objects. Each individual object is described by a tuple (a row of a relation), which contains the set of attribute values unique to that object. The major strength of this model is that it supports a powerful yet simple data manipulation language in which the results of operations on relations are themselves relations. On the other hand, relational databases are weak at handling complex design objects. Design objects typically relate to each other in complex and unstructured ways. In order to fit design objects into a relational model, complex interrelationships and constraints between relations must be built; the end result infringes on the inherent simplicity and uniformity of the relational model. In spite of this limitation, the relational model is still the most powerful among traditional data models. Rasdorf (1982) provides a detailed discussion of the applicability of these data models to structural engineering.

The Entity-Relationship (E-R) model was introduced to add more meaning to the relational data model (Chen 1976). The E-R model is intended primarily for data base design and supports the definition of a data base schema without the concern of other data base issues (e.g., operations on data, physical storage structure, etc.). The E-R data model is based on a perception of the real world which consists of a set of uniquely identified things called entities, and relationships among these entities. Many researchers have built on Chen's work in developing variations of the E-R model.

Object-oriented data models are the most recent additions to the data modelling area. They offer an opportunity to reduce the "semantic gap" between real world entities and their database representations. Its development has been driven by object-oriented languages (e.g., Smalltalk (Goldberg and Robson 1985), C++ (Stroustrup 1986)) and object-oriented data bases, e.g., Gemstone (Maier and Stein 1988), Sembase (King 1986), ORION (Kim 1990). In the object-oriented data models, the basic unit is a *class* of similar *instances* (the actual objects) that defines the *fields* (properties) shared by all instances of the class. *Relationships* can be used to link two instances, two classes, or an instance and a class. Object-oriented data models make use of common abstraction methods such as instantiation/classification, generalization/specialization, aggregation/decomposition, and association.

1.2 Engineering Data Models

Although traditional data models (hierarchical, network and relational) have dominated the business applications area, they have not been able to capture the complexity of engineering data. Several researchers have suggested data models specifically aimed at representing engineering data. These models include the *complex object* model (Lori and Plouff 1983), the *molecular object* model (Batory and Buchmann 1976), and the *functional model* which was suggested in (Shipman 1981) and extended in (Manola and Orenstein 1986). In addition, there are several general purpose data models that have been proposed, including the *Structural* (Wiederhold 1980), *Semantic* (Hammer and McLeod 1981), *Format* (Tsichritzis and Lochovsky 1982), *Binary* (Tsichritzis and Lochovsky 1982), and *Infological* (Sundgren 1974) models. The motivation of these efforts is to provide more flexible data models and to close the "semantic gap" between the real-world and the data model. A number of models have been proposed for structural engineering. These models include Eastman's 1978 model (Eastman 78), the Data Model for Building Design (Law 86), the "Component-Connection" Abstraction Model (Powell et al. 88), etc.

In addition, there has been recent emphasis in engineering data modelling for the purposes of data exchange, engineering views support, and management of the design project life cycle. A close examination at data modelling for engineering in general, and structural engineering in particular, reveals that:

1. Engineering design objects are complex and generally related to each other in different ways through intricate relationships. Most of them have physical, functional, behavioral characteristics about which the designer reasons at one point or another in the design process.

2. During a design process, engineering objects tend to evolve continuously, from one state to another and from one schema to another. This dynamic information growth comes from two directions: new attribute values of the object and further decomposition of the object (Eastman 1978).

3. Several engineering views must be captured for the different data needs among the project participants and across interrelated disciplines of engineering. Each view may refer to different levels of abstraction or aspects of the same design object.

4. Large amounts of heterogeneous data need to be created, maintained, and communicated at various stages throughout the project life cycle.

1.3 Data Exchange Standards

A variety of national and international data exchange standards are being used and developed. The key standards include:

- IGES (1988), Initial Graphics Exchange Specification, is a current US standard for graphical data exchange.

- STEP (IPIM 1988), STandard for the Exchange of Product model data, is a project of the International Standards Organization aimed at capturing and transferring all product data.

- PDES (IPIM 1988), Product Data Exchange using STEP, is the US potential successor to IGES that has evolved into a project to support the development of the STEP standard.

These and the other ongoing standard efforts are primarily focused on defining terminology, data organization, and neutral file formats for batch data exchange. The IGES standard concentrates on representing graphical information; PDES and STEP have a much broader mandate to represent all aspects of product data (where the "product" may be anything from a silicon chip to a high-rise building). Data exchange under these standards is based on a "neutral" data file to which data is written from a post-processor from one software system and from which data is extracted by a pre-processor from another software system. The integrated data models for PDES and STEP use variations of the entity-relationship model to build complex objects that include many different views of data to satisfy all likely generators and consumers of the database.

2. Data Modelling Problem Areas

In developing our approach to data modelling, we have examined a wide variety of data representations for structural and other engineering data. In that process, we have identified a number of problem areas and associated symptoms, including overuse of aggregation hierarchies, nonhomogeneous characterization hierarchies, and replacing relationships with subclasses. The following subsections discuss those specific data modelling problem areas.

2.1 Overuse of Aggregation Hierarchies

Aggregation is an abstraction method that is used extensively in decomposing complex design objects into their ingredient objects. It can also be used in a bottom-up approach to construct complex design objects from the ingredients. Frequently, data models overemphasize the use of aggregation hierarchies in their definitions.

As an example, Figure 1 shows the principal hierarchy from our early work on structural steel framing data (Lavakare and Howard 1989). The first five levels (building through member) are generally appropriate for structural steel, but the emphasis on aggregation is not appropriate for the last three levels (element, part, and connection). For instance, a single steel column (a physical part) may span multiple stories and be conceptualized as multiple members and elements. The relationships between these objects are not necessarily aggregational nor are they necessarily hierarchical.

2.2 Nonhomogeneous Characterization Hierarchies

In characterization hierarchies from a number of data models (e.g., Abdalla 1989, Ito et al. 1990), we observed that many different criteria have been used to distinguish subclasses at the different levels of the hierarchy. In the example shown in Figure 2, the first level of the "component" hierarchy consists of subclasses defined according to their orientation and anticipated structural behavior such as "beams" (horizontal, flexural) and "columns" (vertical, axial and/or flexural). In the second level, the "beams" class has subclasses that are defined according to the material selection: "reinforced concrete" and "steel." The "reinforced concrete beams" subhierarchy includes classes defined according to their cross-sectional shape such as "rectangular" and "T-section," whereas the "steel beams" subhierarchy classifies objects according to the method of fabrication such as "rolled" or "built-up sections." "Built-up steel beams" are further classified according to the connection methods used to build up the part such as "bolted," "welded," and "riveted."

The result is a non-homogeneous hierarchy in which each level involves a different criterion used in defining the object classes. The use of nonhomogeneous characterization hierarchies requires that the data modeller anticipate all possible combinations of characteristics. Consider the sample hierarchy for walls in Figure 3. "Walls" may be "load bearing" or "partition" (separating) or both; they may be located on the "exterior" or "interior," but not both. Note the cross product effect in the hierarchy—the location qualifiers are introduced for every combination of functions. When the number of criteria to be represented in a single hierarchy grows larger, either the resulting data model grows exponentially larger or the data modeller must supply the expertise to eliminate subclasses that represent invalid combinations (and possibly some valid combinations as well).

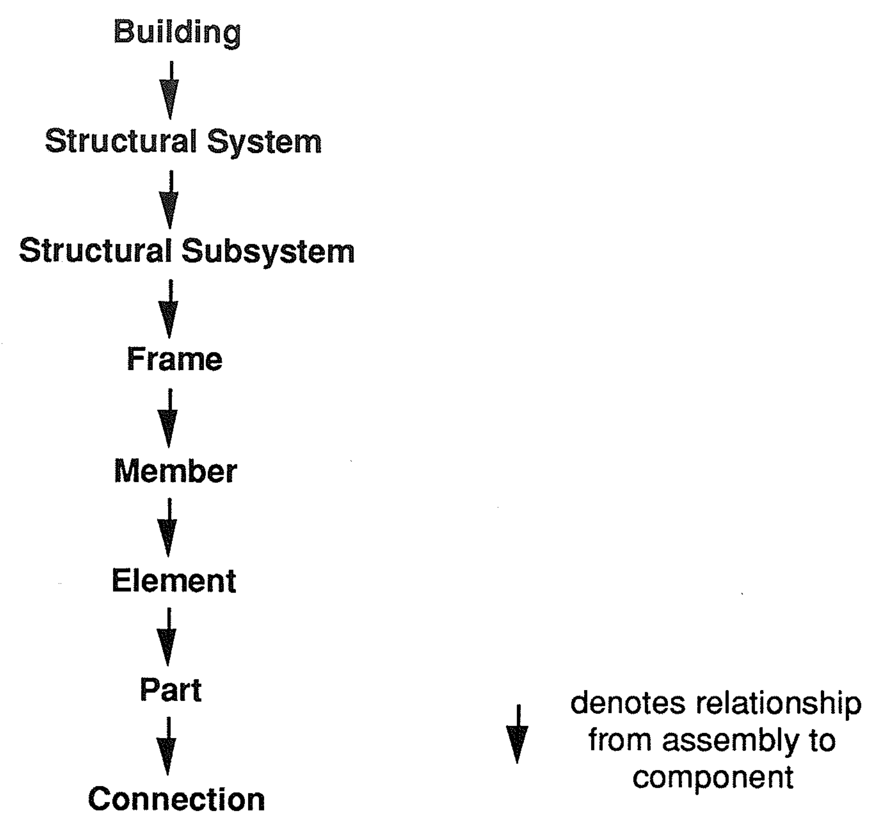

Figure 1: Hierarchy from Structural Steel Framing Data Model (Lavakare and Howard 1989)

2.3 Replacing Relationships with Subclasses

A common problem in data models is that new subclasses are introduced when the nature of the distinction between the subclass and superclass would be better described by a relationship between the superclass and another object. This problem is particularly significant when representing function in structural data models. Consider the two small hierarchies in Figure 4. The subclasses "moment frames" and "braced frames" serve to qualify the superclass "frames," while the subclasses "lateral load resisting" and "gravity load resisting" really characterize the loads supported by the frames. The latter distinction would be better represented by a relationship between the "frames" class and subclasses of the "loads" class. The use of a relationship avoids the redundancy of defining the subclasses in the "loads" hierarchy and the duplication of that classification in the "frame functions" hierarchy to describe all the types of loads that a frame can resist.

A second kind of problem arises when representing the function of components within assemblies. Consider a web stiffener on a steel plate girder. This stiffener *is a* plate that *stiffens* the web of the plate girder assembly. The function of the web stiffener can be derived from its aggregation relationship with the plate girder. That stiffening function may be better represented as a relationship between two objects than as a subclass of the structural function characterization hierarchy.

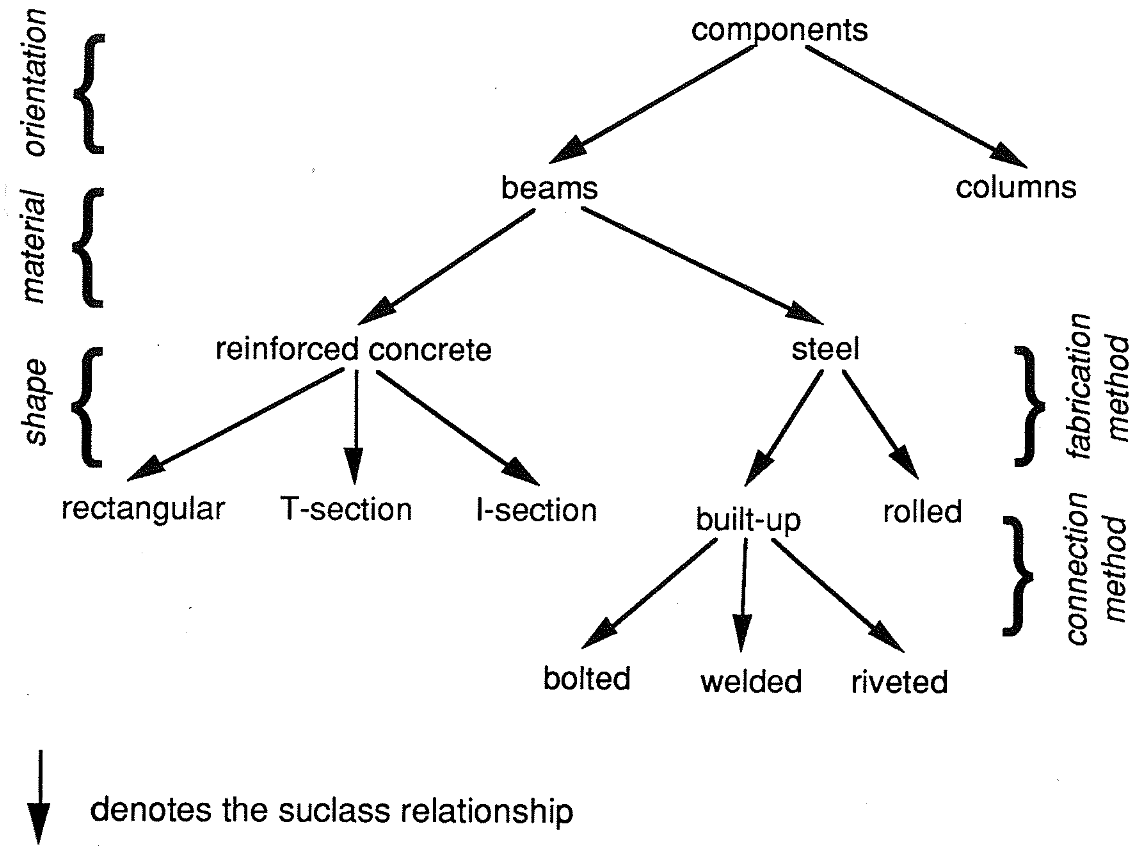

Figure 2: Sample Nonhomogeneous Characterization Hierarchy

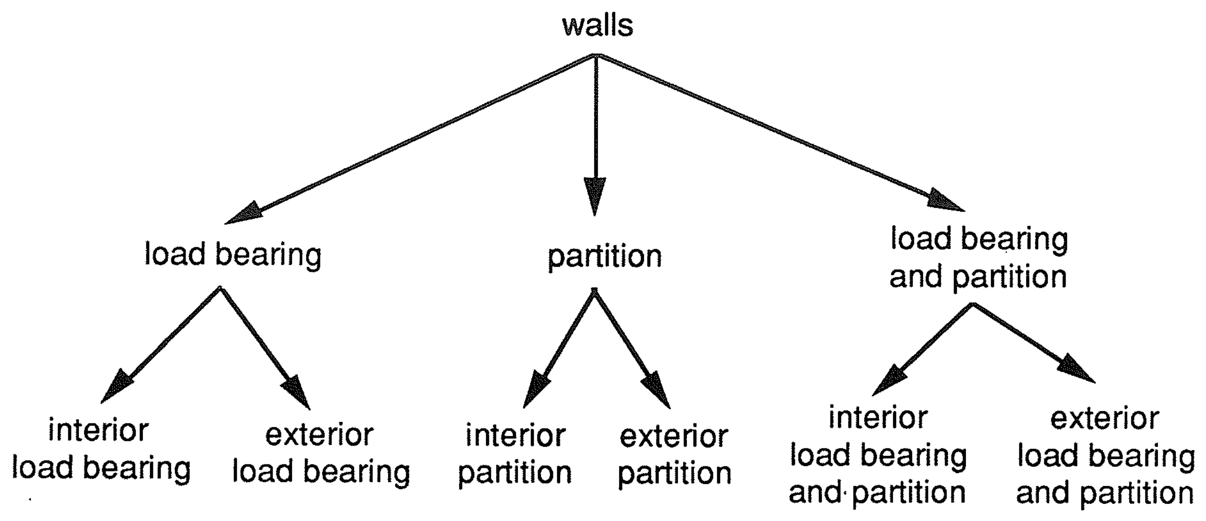

Figure 3: Sample Characterization Hierarchy for Walls

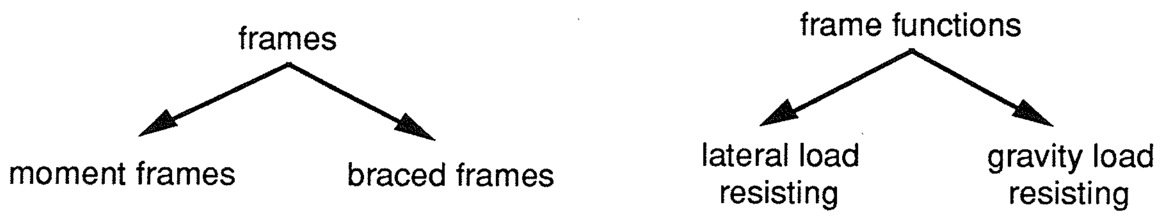

Figure 4: Sample Characterization Hierarchy for Frames

3. A Primitive-Composite Approach to Building Data Models

To support easy exchange of data between engineering computer applications and provide multiple data views while avoiding the data modelling pitfalls identified in the previous section, we have developed the *primitive-composite* approach to building object-oriented models of engineering data. The P-C approach provides a methodology for defining the primitive data concepts of a domain as separate objects and for assembling those concepts to form the composite abstractions that engineers customarily use. By relating composite objects from multiple programs to the primitive concepts, the data in those programs can be shared as primitive concepts derived from one program's composite object and reassembled as another program's composite object without having to provide a direct mapping between those two programs.

This section presents the basic building blocks of the primitive-composite approach, how to apply it to a domain, and how to use it to enable data exchange. However, since there are almost as many different interpretations of object-oriented data models as there are database researchers, we should first present our view of its elements and terminology to serve as a basis for the discussion of our extensions.

3.1. Our View of an Object-Oriented Data Model

The starting point for the primitive-composite approach is the object-oriented data model. The key elements of that model are revisited in items one through five below.

1. **Class** — A class is a descriptive template for a collection of like objects (called "instances" of the class—see below). The class defines descriptive elements (attributes and elements—see below) that are shared by the members of the class. As a convention, we will always name classes with plurals, such as "beams," "walls," and "loads."

In object-oriented programming, the class definition also includes *methods* that describe the behavior of the object. However, that aspect of classes will not be explored further since the initial focus of the primitive-composite data model is on data representation and exchange.

2. **Instance** — An instance (or object) is a unique occurrence of a class. An instance has an identity which uniquely distinguishes it from all other instances. For instance, "beam23" is an instance of the class "beams." An instance does not contain attributes or relationships that are not part of its class.

3. **Attribute** — An attribute contains a value or values that describe the dynamic properties of an instance of the class. The class "beams" might have attributes for "depth," "maximum shear," and "stiffeners required." Each attribute value is associated with a data type (e.g., length, force, boolean). The domain of the data type defines a finite or infinite set of possible values. Data types can be simple types or more complex user-defined abstract data types. Useful abstract data types might include coordinates (a triple of coordinate values), date, time, vector, matrix, etc.

4. **Relationship** — A relationship describes a link between two classes, an instance and a class, or two instances. For example, "beam23" is <u>part of</u> "frame5" and "beam23" is <u>connected to</u> "column16," where <u>part of</u> and <u>connected to</u> are relationships between the indicated class instances. (As a convention, we will always denote relationships by underlining them.) A relationship describes the link in one direction; an inverse relationship describes the link in the

opposite direction. For example, "beam23" is part of "frame5" and "frame5" has subpart "beam23," where part of is the relationship and subpart is the inverse relationship. The definition of a relationship for a class includes a list of the target classes to which that relationship may provide a link. Any instance of a target class or any instance of a subclass of a target class is a candidate value for the relationship (subject, of course, to other constraints).

Three categories of relationships are supported:

- **Characterization** includes all relationships whose definition is based on generalization/specialization or instantiation/classification. The descendents of a general class are specialized by having new attributes, new relationships, or new values for attributes or relationships. Characterization relationships support inheritance from the more general classes to the more specialized ones. Attributes, relationships, and their values may be inherited by specialized classes from the general classes to which they belong. Characterization relationships can link two classes (i.e., generalization/specialization), using the superclass relationship and its inverse subclass; or an instance and a class (i.e., instantiation/classification), using the instance relationship and its inverse instances. For example, "Steels" are a subclass of "materials," and "A36-Steel" is an instance of "steels." (Note: The relationship is-a is frequently used in frame-based and object-oriented programming to convey any or all of these characterization relationships. We will use the more specific terms to avoid confusion.)

- **Aggregation** includes those relationships that relate the component objects to their assembly object. Conversely, aggregation can also be used to decompose an assembly object into its component objects. The basic form of aggregation is represented by the part of relationship and its inverse subparts. However, the aggregation relationship can be used to show more than just assembly and decomposition; it can also describe the function of a component within an assembly. This use of the aggregation relationship is called a *role*, describing the role that a component plays in the assembly. The notion of roles will be very important in representing the function of structural elements. For example: "plate125" stiffens "beam23," where stiffens is a part-of relationship that describes the function of the plate in the beam.

- **Association** is a broad category for relationships that do not represent characterization or aggregation. There are many kinds of association relationships in engineering with specialized semantics such as connected-to, supported-by, adjacent-to, etc.

5. **Schema** — A schema is the set of object class definitions (including attributes and relationships) that defines the data for an application or database. The schema for an analysis program might consist of classes to represent "elements," "nodes," "loads," "stresses," and "displacements."

3.2. Basic Building Blocks of the Primitive-Composite Approach

Items 6 through 13 are the extensions that define the primitive-composite approach to object-oriented data modelling. They describe how the objects are organized to isolate independent concepts and how objects are combined to create complex abstractions that represent user views. Several fundamental rules are included with the basic definitions in order to refine the approach further. Future development of the approach will further expand these rules.

6. **Primitive class** — A primitive class is a class that represents a single concept such as shape, material, or function. It serves as an atomic definition in the data model—something indivisible and basic to the domain.

7. **Primitive characterization hierarchy** — A primitive class hierarchy groups primitive classes that represent increasing specializations of a single concept. As illustrated in Figure 5, two-dimensional shapes can be decomposed into 3-sided shapes, 4-sided shapes, 5-sided shapes, etc. 4-sided shapes can be decomposed into squares, rectangles, trapezoids, etc.

Rule I: (When to Introduce a New Subclass) A subclass in a primitive characterization hierarchy should add at least one attribute or relationship to the set inherited from its super class. This rule requires that the introduction of a new subclass add some new value or link to the description of its instances. If there is no new information, then there is no reason for the subclass to exist. (This rule states the ideal circumstance. Frequently, nonspecialized subclasses will exist as place holders for future attributes, relationships, or knowledge.)

Rule II: (Don't Invent a Class When a Relationship Will Do) A primitive subclass should not be introduced if its only unique feature can be represented by a relationship in its superclass. For example, in a hierarchy of structural functions, the class "supports objects" should not be specialized into "supports static loads" and "supports dynamic loads," but rather the additional meaning should be represented as a relationship supports in the "supports objects" class which can be instantiated as a link to an instance of a "load types" class.

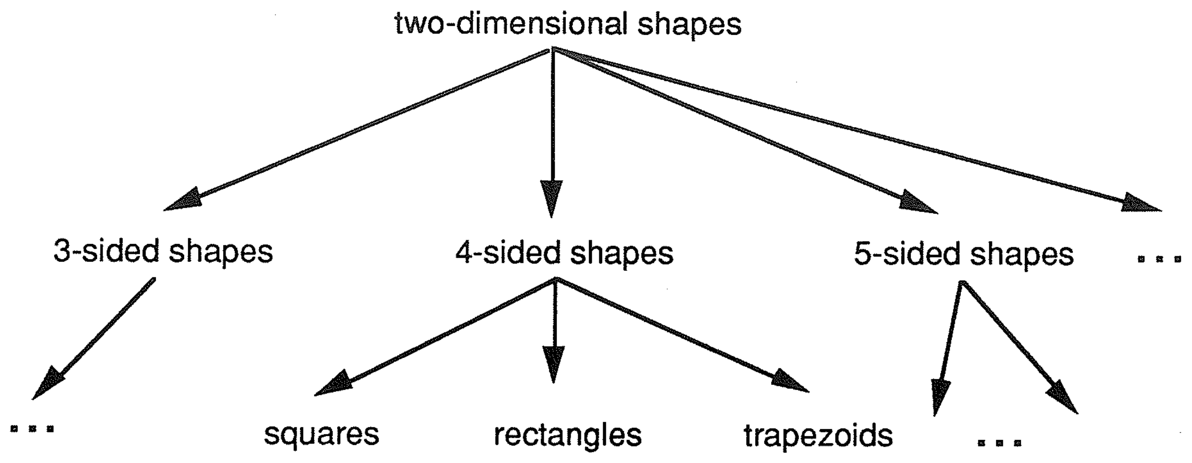

Figure 5: Sample Primitive Characterization Hierarchy

8. **Primitive schema** — A primitive schema is a set of primitive characterization hierarchies that define the basic concepts of an application domain such as structural engineering. The primitive schema defines the concepts that are used directly or indirectly by the domain specialists. Typically, there will be many primitive characterization hierarchies in the primitive schema to cover the breadth of the domain and simultaneously satisfy the requirements that each primitive class represent a single concept.

9. **Primitive database** — A primitive database contains instances of primitive classes from a primitive schema to describe an object or a set of objects. As we will see later, a primitive database can be used as a medium of data exchange between different application systems.

10. **Composite class** — A composite class is a subclass of, aggregation of, or association of two or more primitive classes (see the example in Section 4.2). A composite class describes a complex concept that is defined as a combination of simple atomic concepts in the domain of a given application. For example, the concept of a beam as a combination of specific forms, functions, and behaviors may be built into a program to provide users with abstractions to which they are accustomed. By multiple inheritance, a "Beams" class could be a subclass of more than one physical form class (e.g., for its location, shape, material, etc.), several function classes (e.g., for its load carrying role, load transferring role, etc.), and a behavior class that defines the beam as a flexural element. Similarly, a "Beams" class could relate to different forms, functions and behavior primitive classes by aggregation and association relationships. A composite class may relate to other composite classes as well.

The ability to construct composite classes are an important part of the primitive-composite approach because domain specialists usually reason about objects that represent multiple concepts, combining multiple physical forms, functions, and behaviors into a single object. Using composite classes, an engineer can assemble special purpose objects without being constrained to the list anticipated in a data standard.

Rule III: (No New Attributes in Composite Classes) Ideally, a composite class should not contain attributes or relationships other than those inherited from its superclasses. This rule is really a test for the designers of the primitive schema: does the primitive schema contain all the necessary concepts? and are the concepts organized in such a way that they can be combined to represent all necessary combinations of attributes and relationships that are meaningful in the domain? To state this rule in more realistic terms, the introduction of new attributes and relationships in composite classes should be minimized because the new data would not be exchangeable.

11. **Composite instance** — A composite instance is an instance of a composite class. From the primitive class ancestors of the composite class, the composite instance inherits a set of attributes and relationships that define a new compound object.

12. **Composite schema** — A composite schema is a combination of a primitive schema and a set of composite classes that define an application view of the domain data. Rather than using the primitive classes directly, programmers and database administrators will generally provide their users with composite classes that suit the needs of specialized domain tasks. Composite classes provide a convenient framework for formalizing how the compound abstractions provided in applications are related to the primitive schema.

13. **Composite database** — A composite database is a database that contains instances of composite and primitive classes from a composite schema.

3.3. Building a Primitive Schema

The following algorithm specifies how to build a primitive schema using the primitive approach.

1. Define the classes that belong in the primitive characterization hierarchies.

2. Define the attributes and relationships for the primitive classes.

3. Examine the attributes and relationships in the hierarchies for adherence to Rule I, which states that subclasses must add specifics to superclasses. Correct problems by

cycling through steps 1 and 2 until the superfluous subclasses are satisfactorily justified or eliminated.

4. Test the model by defining composite instances from textbook and real world problems. Repeat steps 1, 2, and 3 until classes from primitive characterization hierarchies can be combined to represent the instances without adding new attributes and relationships specific to the instances.

For complex domains where domain specialists have difficulty isolating concepts, a primitive data model may initially appear to be impossible. Therefore, a composite schema containing a judiciously chosen set of composite classes may be substituted for a completely primitive data model. The key idea is to keep arbitrary combinations of concepts at a minimum.

We want to emphasize that we do not regard the task of building a satisfactory primitive schema as a trivial undertaking. On the contrary, it may prove just as difficult to develop a comprehensive primitive schema as to develop the complex global schema that is the aim of the data standards efforts. However, the flexibility, extensibility, and customizability of the primitive-composite approach should make the resulting primitive schema a more useful artifact.

3.4. Building a Composite Schema

A composite schema may be defined to integrate the data space of an existing application with other related programs or, more ideally, when an application is being developed. The steps in defining a composite schema are:

1. Decide on the application object classes to be included in the model. The set of classes should include all objects in the user view and all internal objects used in the program's reasoning process.

2. Divide the application object classes into those that are completely defined by a single primitive class in the primitive schema and those that represent combinations of primitive classes. For the former set of application object, the task is complete. For the latter set of application objects, define composite classes that are subclasses of the necessary combinations of primitive classes. For convenience, the composite schema may include its own characterization hierarchies of composite classes, with composite objects being defined as subclasses of both other composite classes as well as primitive classes. (A subclass of a composite superclass is by definition a subclass of the primitive classes that comprise the composite superclass.)

3. Examine the set of attributes and relationships inherited from the primitive classes. If some important application values are missing, repeat step 2. Only as a last resort should the composite class introduce attributes or relationships that are not inherited from a primitive class. (In that case, the values for the new attributes and relationships are not exchangeable with other programs.)

4. Test the composite schema with sample problems for the application.

3.5. Using the Primitive-Composite Approach for Data Exchange

The primitive-composite approach supports data exchange by providing a common language for representing concepts in the domain: the . Each application (CAD system, analysis package, database, etc.) is described by a composite schema. Comparison of the composite schemata of two applications will immediately reveal overlapping data coverage simply by locating the primitive classes referenced by both composite models. When two applications need to exchange data, the data from the first application is translated into a primitive database (containing only instances of primitive classes) and then translated into the composite form of the second application.

Furthermore, the two applications do not need to share the composite classes. They need only share primitive classes from which their composite classes are formed. Figure 6 shows four composite schemata which correspond to different applications, but are related to one primitive schema. The exchangeable data is identified by the primitive classes which are referenced by both composite models. For example, an architect may reason about a wall in terms of its location, dimensions, color, and texture. A structural engineer will need to share the data about location and dimensions, but will add data about materials, loads, and stresses. The building contractor will use the engineer's data and add data about costs and scheduling. The interior finishing subcontractor will go back to the architect's original data to order paint and plaster. The primitive schema contains the primitive classes necessary to assemble the four different composite wall classes and defines how overlapping data may be exchanged between those composite classes.

This style of neutral file exchange is commonly used in current data exchange methodologies. The difference with the primitive-composite approach that the translations can be automatically inferred from the information in the applications' composite databases. The data exchange can also be dynamic, using a framework such as KADBASE (Howard and Rehak 1989) to support real-time data queries between heterogeneous applications. (KADBASE is a flexible, knowledge-based interface in which multiple knowledge-based systems and multiple databases can communicate as independent, self-descriptive components within an integrated, distributed engineering computing system.)

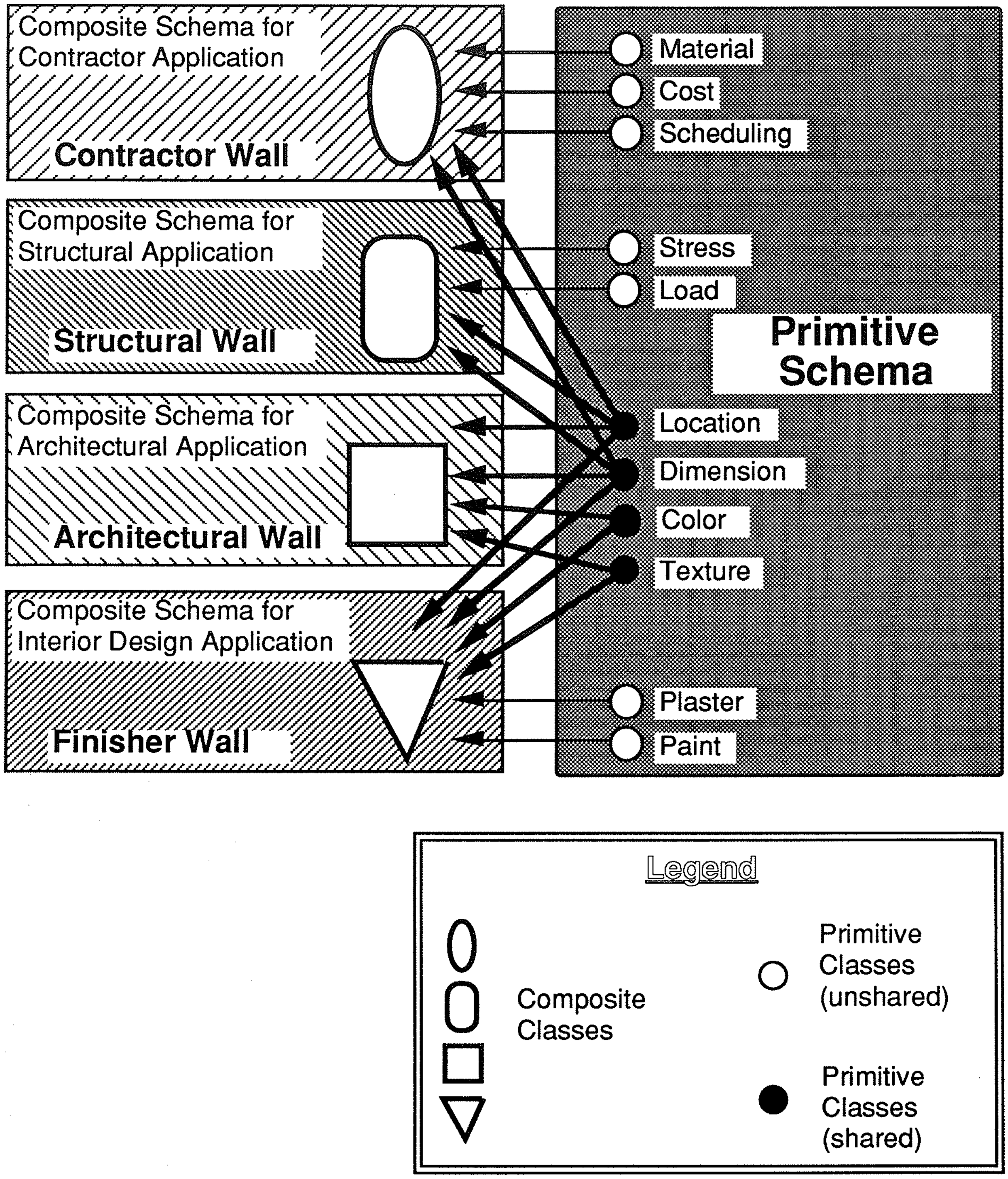
Composite Schema for Contractor Application
Contractor Wall
Composite Schema for Structural Application
Structural Wall
Composite Schema for Architectural Application
Architectural Wall
Composite Schema for Interior Design Application
Finisher Wall
Material
Cost
Scheduling
Stress
Load
Primitive Schema
Location
Dimension
Color
Texture
Plaster
Paint
Legend
Composite Classes
Primitive Classes (unshared)
Primitive Classes (shared)

Figure 6: Data Integration Through the Primitive Schema

4. Application to Structural Engineering

In this section, we examine the application of the primitive-composite approach to structural engineering data modelling. *Form, function*, and *behavior* are the basic conceptual building blocks for defining the primitive characterization hierarchies. Using the primitive-composite approach, we can define common structural design objects as composite objects in terms of form, function, and behavior primitives selected from different characterization hierarchies. Luth (1991), Umeda et al. (1990), and Garrett and Yau (1989) have studied form, function, and behavior in the representation of structural and other engineering objects, with different emphasis.

Our purpose in this section is to identify the considerations and issues in defining the elements of form, function, and behavior necessary for structural design objects, and to briefly demonstrate how a simple beam object might be defined using the elements of a primitive structural data model. The detailed development of a primitive-composite schema for structural engineering is the subject of our ongoing research.

4.1 Form, Function, Behavior

Form, function, and behavior provide the three principal axes in the definition of a primitive-composite structural data model. In Figure 7, different aspects of form, function and behavior are shown as the three orthogonal planes that compose the complete description of complex design objects. Figure 8 provides a simple example of a load-bearing wall supporting beams and a slab for use in the explanations in this section.

4.1.1 Form

Form descriptions of an object define its physical characteristics. There are many types of form description: spatial, geometric, topological, material, fabrication features, etc. This section covers those aspects of form necessary to define structural engineering objects. (Many elements of the form description are within the STEP data exchange specification and can be adapted to fit within the primitive-composite approach. We will reference some of the useful STEP elements as we proceed.)

The *spatial form* of an object describes the spatial envelope of the object as well as its location and its orientation in three-dimensional space with respect to a global reference point or relative to other objects in its environment. The spatial envelope of a physical object can be defined in terms of a local coordinate system and the dimensions (length, width, and height) of its spatial enclosure.

For example, the wall object can be located and oriented by its local coordinate system or in reference to its neighboring objects such as the foundation or the floor slab.

The *geometric form* of an object defines shape. Physical objects are three-dimensional, but their shapes can be represented by different geometric forms. Consider the wall object again. First, it can be represented with a cross section on a planar surface and a line segment in the third dimension. Second, it can also be represented by a cross-section and a projected view in the third dimension. This method of using two fundamental views (front, top, or side view) to construct the geometry of an engineering part is commonly used in engineering drawings. Finally, the wall object can be represented as a solid parallelopiped. Three-dimensional shape models (such as solid, surface, and wire frame models) are defined in the STEP Integrated Product Information Model (IPIM 1988).

The *topological forms* define the connectivity of objects in the constructed environment. In structural engineering, a wire frame model of the structure is commonly constructed in order to define the topology of the structure. The wire frame models are also analogous to finite element models used for structural analysis purpose. Vertices (dimensionality 0), edges (dimensionality 1), faces (dimensionality 2), volumes (dimensionality 3) are topological primitive entities whose standard definition is available from the STEP Integrated Product Information Model and the GARM model (Gielingh 1988).

The *material form* of an object describes the type and properties of the material that comprises the object. The material types used in civil engineering include steel, reinforced concrete, asphalt, mortar, timber, etc. In reference to the STEP Integrated Product Information Model, material properties can be classified into groups such as physical, structural, thermal, and thermal expansion. The material property primitives in these groups can be defined in separate characterization hierarchies and used to describe isotropic, 2-dimensional anisotropic and 3-dimensional anisotropic materials.

The *fabrication form* includes features of an engineering part that the designer prescribes for building the part. There is a large set of standard fabrication features such as taper, bend, thread, cut out hole, edge clipping, edge preparation, NC mark, etc. These standard fabrication features are defined in the STEP Integrated Product Information Model and the NIDDESC Ship Structural Model (NIDDESC 1988).

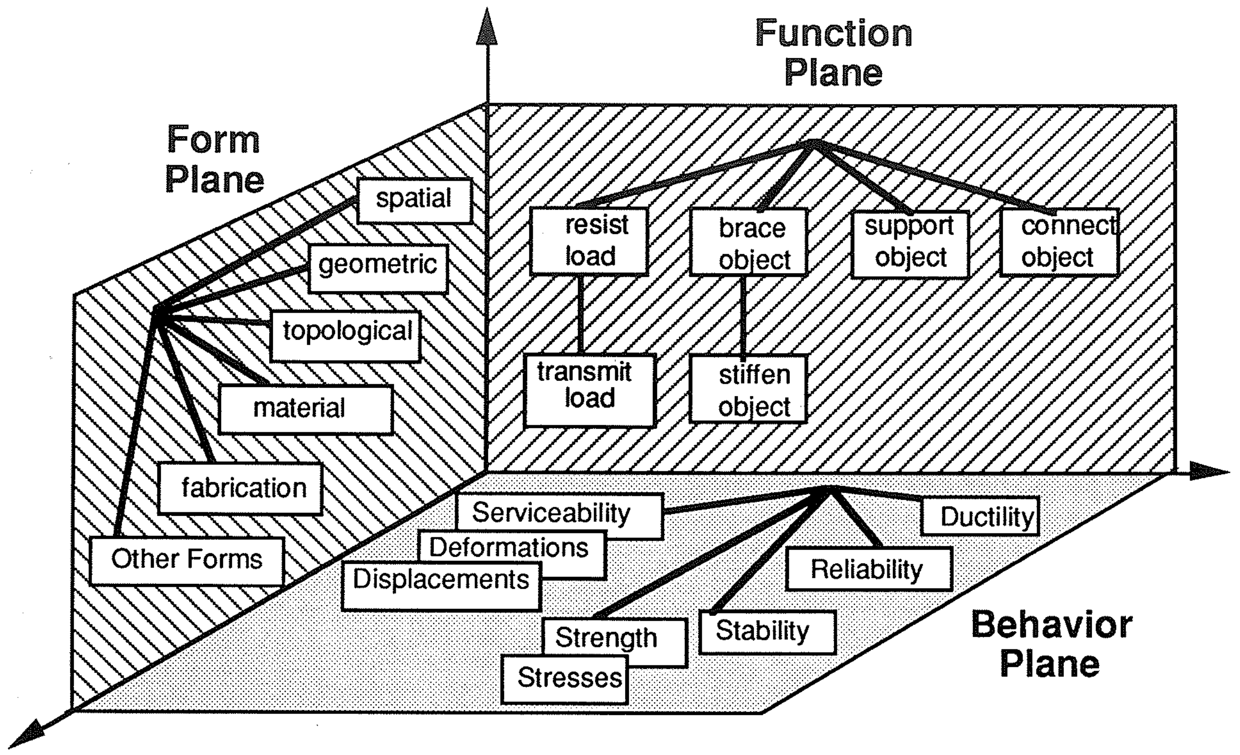

Figure 7: Form, Function, and Behavior for Structural Engineering

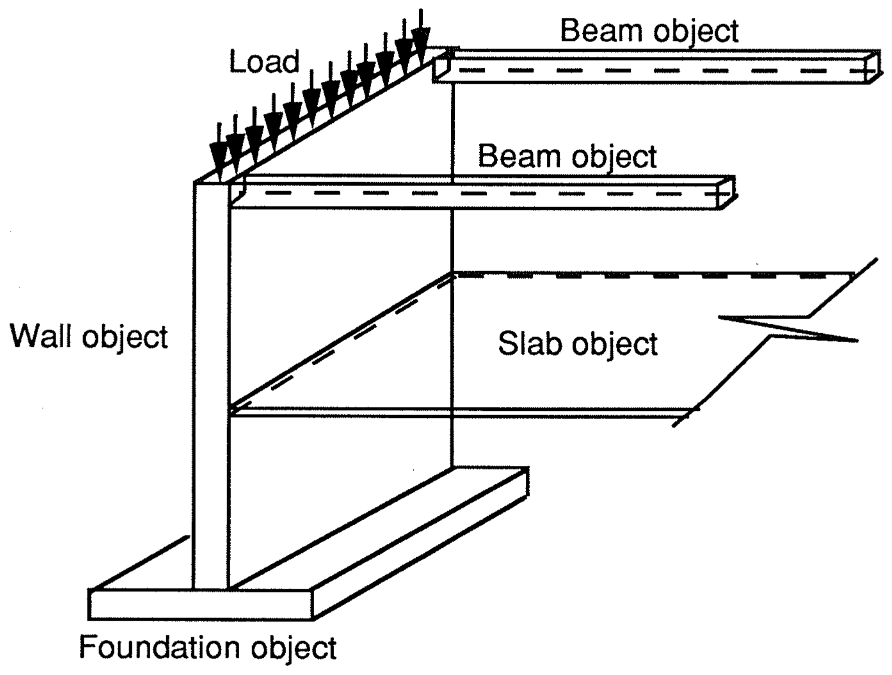

Figure 8: Example of a Load-Bearing Wall Object

4.1.2 Function

The functional description of an object reflects the intended role or purpose of the object in its constructed environment. An object may serve several functions. The wall object in Figure 8 may resist loading (structural function) and provide a partition (architectural function). Even within one view such as structural engineering view, the wall may perform several functions such as *resist* loads (gravity or lateral), *support* the girders and floor slab, and *transfer* the applied loading to the foundation.

From the viewpoint of structural data modelling, functions of the building elements relate to some aspect of load carrying, load transferring, member or part connecting, member supporting roles, and so forth. The functions are frequently captured as specialized relationships between the building elements and other elements, or between the building elements and other entities in the model that takes part in the functional definition such as loads, load cases, etc. We have identified the following key roles of common structural objects:

1. Load resisting function
2. Load transmitting function
3. Object supporting function
4. Object connecting function
5. Object bracing function.

These functions are illustrated in the function plane in Figure 7. Although there may be other noteworthy functions, we are focusing our initial study on these functions.

The *load resisting function* of an object is to withstand a load (or loads) that is applied directly to it or transmitted from another object. The wall object in Figure 8 resists its own weight, the external loads directly applied to it, and the loads transmitted from the connected beams and floor slab. The *load transmitting function* of an object is to transfer the loading it carries to other objects in the load path. For example, the wall object in Figure 8 transmits its loads to the foundation. The *object supporting function* is to support another object. This function enables the transfer of loads from the supported object, to the supporting object, down to the next object in the load path. The wall object in Figure 8 supports two beam objects and the floor slab object. The *object connecting function* is to connect two or more objects together. For example, the function of a beam-to-wall connection object is to connect the beam object to the wall object. The *object bracing function* is to brace another object.

4.1.3 Behavior

The behavior of a design object is the way that object responds to environmental stimuli in carrying out a certain function. Since an object may perform several functions, it follows that it may exhibit different behaviors, each of which corresponds to the particular function in question. For instance, in resisting gravity loads, the wall develops internal axial stresses; in resisting lateral loads, it exhibits shear and bending stresses.

In structural engineering, the behavior of a structural component under the influence of loading is manifested in terms of internal forces and stresses, deflection, deformation, vibration, etc. Generally speaking, the design of a structural component has criteria that impose limits on its behavior. These design criteria ensure the acceptable performance of the design object according to professional standards from the following perspectives:

1. Strength (stresses and internal forces)
2. Serviceability (deflection, vibration, cracking, etc.)
3. Ductility
4. Stability
5. Reliability.

4.2 An Integrated Example of Form, Function, Behavior

In this section, we show an integrated example of form, function, and behavior in defining a composite class for beams. Consider a simply supported, 10-foot beam that carries a distributed live load w_L of 10 kip/feet as shown in Figure 9. In addition to the live load, this beam also supports a piece of mechanical equipment. This results in a concentrated load P_D of 20 kips in the middle of its span. The beam is an A36 steel standard rolled steel shape W14x30. Figure 9 shows a composite instance of a composite class. This example is used in the following section to demonstrate how a composite class can be defined in terms of its primitive classes using the aspects of form, function and behavior.

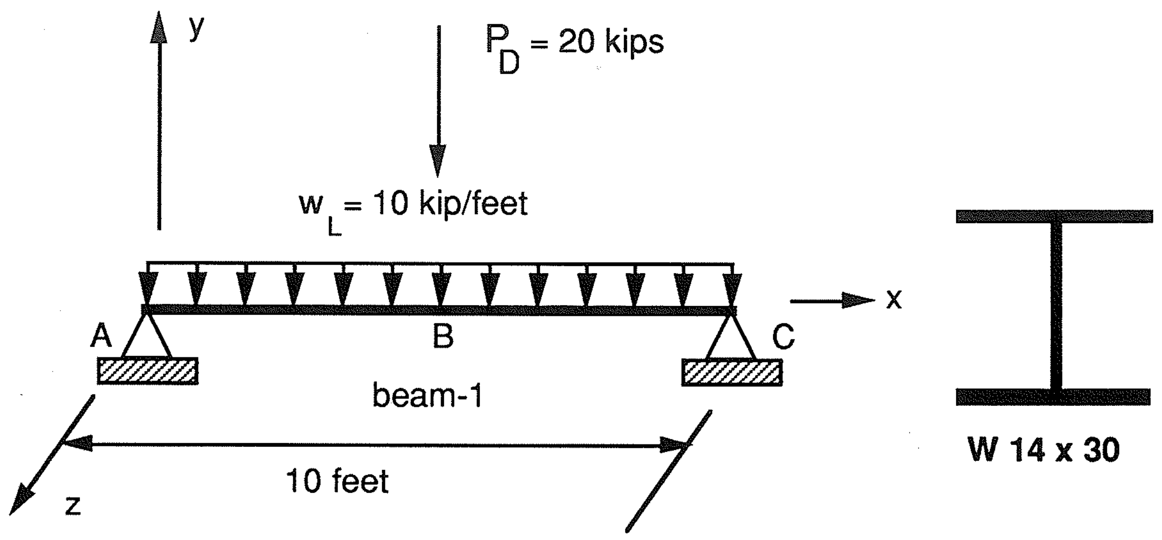

Figure 9: Simply Supported Beam Example

The definition of a composite class "Beams" includes several links with various primitive classes from the form, function, and behavior class hierarchies that describe these aspects of a beam. These links represent different relationships as shown in Figure 10. The primitive classes that comprise the composite class include:

- a primitive class from the spatial form hierarchy that includes attributes to position the beam in space, including the coordinates of end A, the coordinates of end B, and the length of 10 feet.
- a primitive class from the geometric form hierarchy representing the uniform cross-section with the shape of a standard wide-flanged section (W14x30).
- a primitive class from the topological form hierarchy that establishes a connection between the beam and the supports at A and C.
- a primitive class from the material form hierarchy containing the properties for A36 steel.
- the "load-resisting" primitive class from the function hierarchy linking the beam with the distributed dead load and the concentrated equipment load.

- a flexural response primitive class from the behavior hierarchy, which describes the behavioral responses of the beam due to its loading such as internal forces, degrees of freedom, boundary conditions, stresses, deformations, displacements, etc.

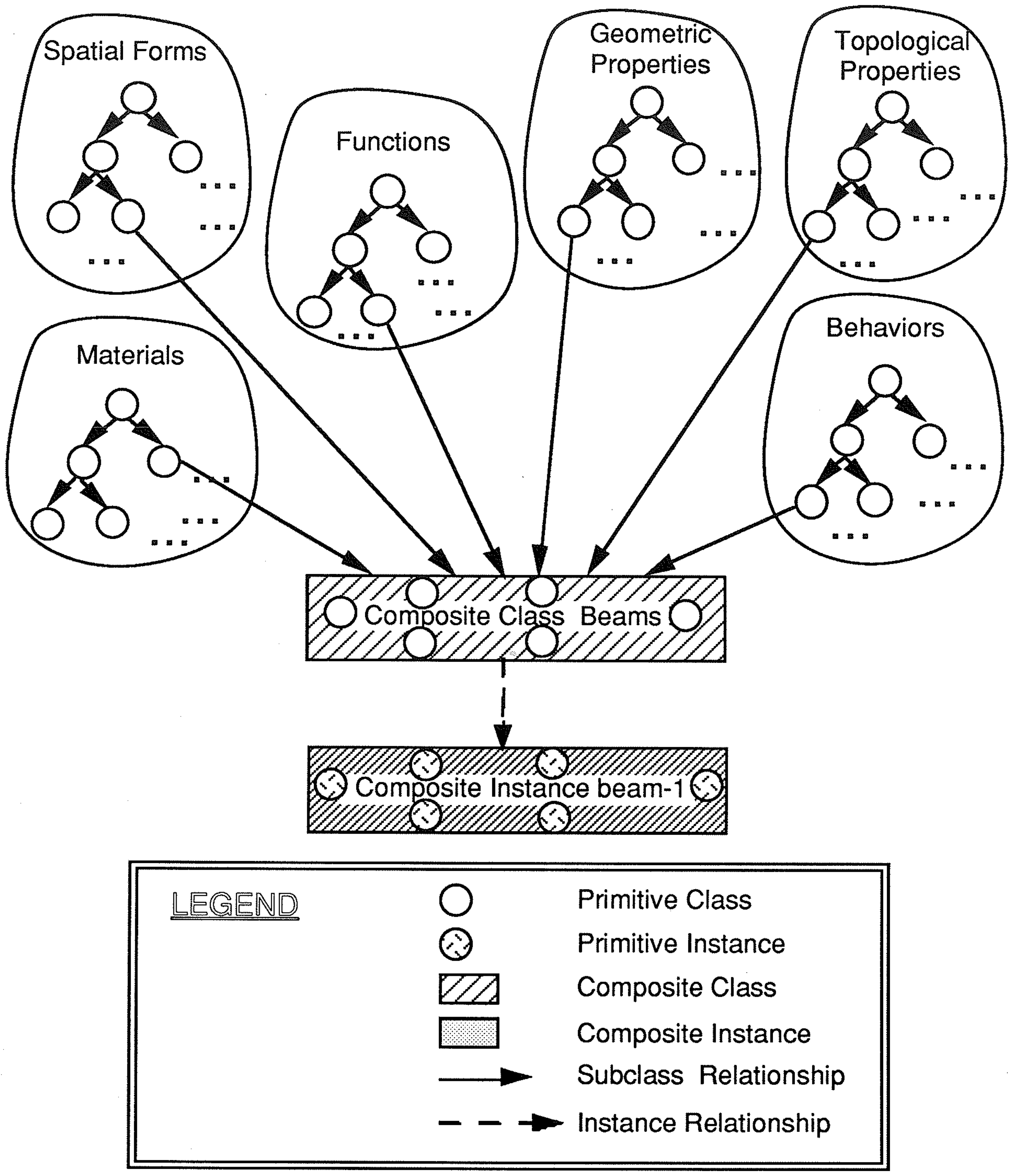

Figure 10: An Composite Instance Representing a Beam in Terms of Its Primitives

5. Conclusions on the Primitive-Composite Approach

Using the primitive-composite approach, it is possible to develop a *language* for domain data through the primitive characterization hierarchies of the primitive schema. To extend the analogy, computer tools and users may speak specialized *dialects*, which are combinations of elements found in the common language. Translation from one tool to another is driven by the mapping information that links the composite schemata (dialects) to the primitive schema. It is important to note that the primitive-composite approach is focused on the logical translation of data and does not address issues of data ownership, integrity, consistency, and currency. However, without a solution for the logical data translation, the other issues are just abstract questions.

The primitive-composite approach has the potential to produce the following impacts on engineering data modelling and data exchange:

- **Building complex integrated data models to support data exchange is no longer necessary.** Rather than trying to anticipate every possible combination of engineering data, data exchange models need only define the relatively stable set of primitives of the domain. The primitive-composite approach provides the methodology for building composite objects as needed by users and programs as well as the mechanism for translating those composite objects from one system to another via the primitive schema. Users and programs need not share the same composite schema to exchange data that they have in common.

- **Complex user views are easy to develop.** Using the primitive-composite approach, the elements of form, function, and behavior are separated in disjoint primitive characterization hierarchies. Each primitive characterization hierarchy uses only one homogeneous criterion to define the primitive classes and therefore provides a clean view about a specific aspect of a complex engineering object description. The strategy of view separation in defining a complex design object enables the developer or different developers to concentrate on one particular view of the object at a time. Moreover, this "divide and conquer" modelling technique facilitates the creation of the model and the abstraction of different views.

- **Many user views can be supported from the same primitive schema.** The possibility of defining new composite object types by identifying their primitive constituents provides the user with a great deal of modelling flexibility. This approach allows users and programmers to represent a wide array of different views about complex objects and to capture their informational evolution throughout the design stages and the project life cycle.

- **Programs can be self-descriptive.** A program with a formally defined composite schema essentially carries its own descriptive knowledge to support the exchange of common data with any other program that shares the same primitive schema. Programmers do not need to build knowledge about an application's data into a special purpose translator—all of that knowledge is already present in the definition of the composite schema.

- **Software developers can add value through customization without restricting data exchange.** One pitfall of a standard is that it reduces the ability of vendors to differentiate their products. In the P-C approach, even though the programs must be built upon the primitive schema, developers can still produce complex and highly customized applications without sacrificing easy data exchange. To provide the data exchange capability, the developer need only define how the composite schema of the system is assembled from the primitive schema. The tool and the data can be optimized to solve the problem at hand, while retaining the ability to easily exchange data with other systems that have overlapping data.

- **Domain knowledge bases can be shared if they reference the primitive schema.** The most common knowledge bases of domain information for structural engineering are the requirements from building codes and standards. Research in computerized standards processing has shown that a critical prerequisite to the encoding of the requirements is formalize the data model implicitly represented within the standard (Garrett and Fenves 87). If the requirements address the data in the terms of the primitive schema (e.g., a combination of forms, function, and behavior primitives), then those requirements can be used for checking or design in any composite model. In the same way, other knowledge bases (both general and specific) can be shared if they reference the primitive schema.

Our continuing research in this area is refining the basic concepts of the P-C approach, defining a comprehensive primitive schema for structural steel, developing software tools to support the assembly of primitive and composite schemata, and exploring how the data translation using the primitive-composite can be automated within the KADBASE framework.

6. Acknowledgments

This work has been supported by a grant from the Center for Integrated Facility Engineering at Stanford University. The authors extend their thanks to Jared Nedzel, Sanjai Tiwari, and especially Ajay Lavakare, who have contributed ideas to our ongoing work in the area of data modelling.

7. References

Abdalla, J. A. (1989). *Object-Oriented Principles and Techniques for Computer Integrated Design.* Unpublished Ph.D. Thesis, Department of Civil Engineering, University of California at Berkeley, Berkeley, CA.

Batory, D. S. and Buchmann, A. (1976). "Molecular Objects, Abstract Data Types and Data Models: A Framework." *Proceedings of the Conference on Very Large Data Bases*, 1(1), 9-36.

Chen, P. P. S. (1976). "The Entity-Relationship Model - Toward a Unified View of Data." *ACM Transactions on Database Systems*, 1(1), 9-36.

CODASYL Systems Committee (1971). *Feature Analysis of Generalized Data Base Management Systems*. Technical Report.

Codd, E. F. (1970). "A Relational Model for Large Shared Databanks." *Communications of the ACM*, 13(6), 377-390.

Date, C. J. (1990). *An Introduction to Database Systems.* Vol. 1, 5th ed., Addison-Wesley, Reading, MA.

Date, C. J. (1983). *An Introduction to Database Systems.* Vol. II, Addison-Wesley, Reading, MA.

Eastman, C. M. (1978). "The Representation of Design Problems and Maintenance of their Structure." *Artificial Intelligence and Pattern Recognition in Computer-Aided Design*, North Holland, 335-363

Garrett, Jr., J. H., and Fenves, S. J. (1987). "A Knowledge-Based Standard Processor for Structural Component Design." *Engineering with Computers*, 2(4), 219-238.

Garrett, Jr., J. H., and Yau, N. J. (1989) "Issues in Representing Engineering Design Decisions for Support of Concurrent Engineering." *MIT-JSME Workshop on Concurrent Engineering,* Boston, MA.

Gielingh, W. (1988). *General AEC Reference Model(GARM): an aid for the integration of application specific product definition models.* Unpublished PDES/STEP working document.

Goldberg, A. and Robson, D. (1985). *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA.

Hammer, M., and McLeod, D. (1981). "Data base description with SDM: A semantic database model." *ACM Transactions on Database System*, 6(3).

Howard, H. C., and Rehak, H. C. (1989). "KADBASE: A Prototype Expert System-Database Interface for Engineering Systems." *IEEE Expert*, 4(3), 65-76.

Ito, K., Law, K., and Levitt, R. (1990). "PMAPM: An Object-Oriented Product Model for A/E/C Process with Multiple Views." CIFE Technical Report No. 34, Stanford University, Stanford, CA.

IGES: Initial Graphics Exchange Specification (1988). Version 4.0, U.S. Department of Commerce, National Bureau of Standards, National Engineering Laboratory, Center for Manufacturing Engineering, Automated Production Technology Division, Washington, D.C.

IPIM: Integrated Product Information Model (1988). Working Draft, Document No. 4.1.2, PDES/STEP, International Standards Organization TC184/SC4/WG1.

Kim, W. (1990). "Object-Oriented Databases: Definition and Research Directions." *IEEE Transactions on Knowledge and Data Engineering*." ACM, 2(3), 327-340.

King, R. (1986). "A Database Management System Based on an Object-Oriented Model." *Expert Database Systems*, Benjamin/Cummings, Menlo Park, CA, 443-468.

Lavakare, A. and Howard, H. C. (1989). "Structural Steel Framing Data Model." CIFE Technical Report No. 012, Center for Integrated Facility Engineering, Stanford University, Stanford, CA.

Law, K. H., and Jouaneh, M. K. (1986). "Data Modelling for Building Design." *Computing in Civil Engineering, Proceedings of the Fourth Conference*, ASCE, 21-36.

Lorie, R and Plouff, W. (1983). "Complex Objects and their Use in Design Transactions." *Proceedings for Database Engineering Applications*, Database Week ACM, 115-121.

Luth, G. P. (1991). *Representation and Reasoning for Integrated Structural Design of High-Rise Commercial Office Buildings*. Forthcoming unpublished Ph.D. thesis, Department of Civil Engineering, Stanford University, Stanford, CA.

Maier, D., and Stein, J. (1988). "Development and Implementation of an Object-Oriented DBMS." *Research Directions in O-O Programming*." B. Shriver and P. Wagner, eds., MIT Press, Canbridge, MA, 355-392.

Manola, F., and Orenstein, J. A (1986). "Toward a General Spatial Data Model for an Object-Oriented DBMS." *Proceedings of the 12th International conference on Very Large Data Bases*, Kyoto, Japan, 328-335.

NIDDESC: Navy/Industry Digital Data Exchange Standards Committee (1988). *Reference Model For Ship Structural Systems*, Version 3.0, International Standards Organization TC184/SC4/WG1 Document 3.2.2.5.

Powell, G., et al. (1988). "A Database Concept for Computer Integrated Structural Engineering Design." *ASCE Fifth Conference on Computers*, Alexandria, VA, 521-529.

Rasdorf, W. J. (1982). *Structure and Integrity of a Structural Engineering Database*, Technical Report DRC-02-14-82, Design Research Center, Carnegie-Mellon University, Pittsburgh, PA.

Shipman, D. W. (1981). "The Functional Data Model and the Data Languages DAPLEX." *ACM Transactions on Database Systems*, 6(1), 140-173.

Stroustrup, B. (1986). *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts.

Sundgren, B. (1974). "The Infological Approach to Data Bases." *Data Base Management*: North-Holland.

Tsichritzis, D. C., and Lochovsky, F. H. (1982) *Data Models*, Prentice-Hall, Englewood Cliffs, N.J.

Umeda, Y., Takeda, H., Tomiyama, T., and Yoshikawa, H. (1990). "Function, Behavior, and Structure." *Applications of Artificial Intelligence in Engineering V, Proceedings of the Fifth International Conference*, Vol. 1, J.S. Gero, ed., Boston, MA, 177-194.

Wiederhold, G., and ElMasri, R. (1980). "The Structural Model for Database Design." *Entity-Relationship Approach to System Analysis and Design.* North-Holland, 237-257.