An Approach to Modeling Database Activity

Technical Report 239

by
Randell Sherman Flint

October 24, 1984

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Information and Computer Science,
University of California, Irvine

Committee in charge:
Professor Nancy G. Leveson
Professor Peter A. Freeman
Professor Rami R. Razouk

Table of Contents

List of Tables

List of Definitions

An Approach to Modeling Database Activity

Technical Report 239

by

Randell Sherman Flint

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1984

Professor Nancy G. Leveson, Chair

Results in the field of data modeling currently suffer from many of the same ills which plagued data management systems in the late 1960's. Advanced semantic modeling systems such as the *Semantic Data Model* and the *Relational Model/Tasmania* are extremely complex to understand as well as somewhat *ad hoc* in design. Such systems capture only *static snapshots* of activity in the world being modeled. On the other hand, behavioral models which do attempt to model system dynamics typically provide less overall modeling power than comprehensive semantic models. Further, the specifications of behavior which can be expressed with such models are themselves static snapshots which are not integrated with other database objects.

This work describes one approach for capturing dynamic relationships by distilling the concepts found in semantic and behavioral data models into a small number of flexible constructs. The resulting *Prototype Activity Modeling System* (*PAMS*) captures the containment, feedback, operational, and state dependency roles of entities in the world being modeled. Further, these definitions of database activity are captured as database objects (rather than as a schema) so as to allow dynamic manipulation of entity roles.

The key concept of the approach is the *bundle* — a purposefully designed extension of time-proven relational database modeling concepts which includes support for presentation ordering and complex Cartesian aggregations. By applying the basic *nested bundle principle*, it is possible to obtain complex hierarchies of static structural information. The static templates so constructed, when used with a non-procedural query language and the *value nomination principle* which reduces relations to scalar values when necessary, provide a conventional database modeling system for applications. By extending these templates with the *non-procedural thunk principle* which embeds query specifications within object definitions, variations caused by dependencies within the application can cause the apparent contents of the database description to change. When further extended by the *activity monitoring principle* which records the interaction between the application and its environment, these dynamic templates can account for changes outside the scope of the application.

CHAPTER 1

Introductory Premise

Over the years research efforts in the field of database management have progressed through several levels of abstraction. The late 1960's were characterized by efforts to define just what constitutes a *database management system.* As the 1970's arrived, so did the era of the *data model* as evidenced by the development of Codd's *Relational Model* [Codd1970], CODASYL's *DBTG Network Model* [CODASYL1971], and the initial publication of the ANSI/X3/SPARC database management system framework model [ANSI1975]. During the later portion of the 1970's, most research concerning data models involved drives to increase the "robustness" of models by including ever larger amounts of semantic information about the data. These *semantic data models* are well represented by Smith and Smith's work on *Aggregation* and *Generalization* [Smith1977a, Smith1977b], Hammer and McLeod's *Semantic Data Model* (*SDM*) [Hammer1978], and Codd's *Relational Model/Tasmania* (*RM/T*) [Codd1979].

Codd's work, in particular, has had an undeniably major impact on all research in data modeling since the Relational Model broke significant new ground. While the point is certainly arguable (and is further addressed in Chapter 2 of this work), Codd's RM/T is probably the most completely developed semantic data model. However, nearly five years have passed since its widespread dissemination and during that time there has been little interest concerning its further development. More likely than not, the reason for this lack of further development can be traced to its complexity. While quite robust, the model is also complex enough that thorough study is required before it can be used by anyone [Codd1979, p. 432]:

> It should be remembered . . . that the extensions in RM/T are primarily intended for the minority consisting of database designers and sophisticated users; most users will probably prefer the simplicity of the basic Relational Model.

Unfortunately, similar complexity problems seem to plague all semantic data models. The objective of the research reported on herein is to produce a modeling system which has both the representation power of a semantic data model and the "friendliness" of the Relational Model.

1.1. The Concept of Database Activity

The original motivation behind this work was simply a desire to understand what makes semantic data models such as RM/T and SDM so complex. The first and most obvious step toward an answer is an understanding of what such models capture in the way of "semantics." Interestingly, these models do not attempt to define the scope of what they can model. Generally, each approach appears as a collection of "cookbook" techniques for modeling specific types of semantics with no attempt to place the techniques within any well defined whole. Thus, comparing such models is often like comparing apples with oranges, though from the inside out since the skin of the model is not represented by any singular concept. In this regard, Codd notes that the term *semantic data model* is somewhat misleading [Codd1979, p. 398]:

> Actually, the task of capturing the meaning of data is a never-ending one. So the label "semantic" must not be interpreted in any absolute sense. Moreover, database models developed earlier (and sometimes attacked as "syntactic") were not devoid of semantic features (take domains, keys, and functional dependence, for example). The goal is nevertheless an extremely important one because even small successes can bring understanding and order into the field of database design.

In other words, the modeling of data semantics is really a continuum from some very simplistic notions to concepts probably far beyond those incorporated in present semantic data models.

As an attempt to introduce a degree of order to the discussion of database semantics, the phrase *database activity* shall be used herein to denote a particular fixed point along the continuum. For the moment, this new term shall be defined as simply *whatever semantics are described by contemporary semantic data models* though a much more precise definition will evolve in the next chapter. (While the author will admit that the introduction of new terms to describe old concepts is clearly undesirable, the choice seems justified in order precisely to identify the classes of semantics handled by contemporary models.)

1.2. Developing a Model of Database Activity

Given that this work attempts both to *define* the concept of database activity and to *develop a model* for the concept, it is quite possible that the results will be less than optimal if either the definition or the model proves to be an inaccurate representation of reality. Such may be the case if for no other reason than the far-reaching nature of the task at hand. Thus a caveat seems in order; the results of this work are governed by several hypotheses:

(1) The description of a database is just as much a "real world entity" as anything else which is to be represented abstractly in a database. Thus, the database description is just as much a subject for data modeling as is the database itself.

(2) The data in a database undergoes change for any number of reasons (as discussed later); any realistic model of a database must account for such change.

(3) Given a choice between two data models (or two data modeling concepts) of similar scope, the "smaller" of the two is better. This means that flexibility should be prefered: a small number of concepts connectable in a large number of ways is more desirable than a large number of concepts connectable in only a few ways.

(4) Even though it is considered by many to be the most flexible of the simple data models, the Relational Model is too restrictive to capture the information required to model database activity.

Where appropriate, the rationale behind these ideas is discussed in ensuing chapters as a powerful semantic data modeling system is developed.

Chapter 2 pursues the more detailed definition of *database activity* as an integrated view of current semantic data modeling concepts. Both the scope and applicability of the concept are compared to those found in commonly used modeling methodologies as well as in other modeling research efforts.

Chapter 3 considers the question of how to express any given description of database activity with the conclusion that a database (of the description) is most appropriate. An examination of the Relational Model, its limitations, and the foundations which underlie relations leads to the structural notion of a *bundle* which extends relational modeling concepts in

ways useful for representing database activity.

Chapter 4 uses the bundle concept to develop a particular model of database activity called the *Prototype Activity Modeling System* (*PAMS*) by examining the semantic dependencies found in relationships among objects. The system is beyond Codd's RM/T in expressive power, yet due to its unified nature is minimally more complex than the original Relational Model.

Chapter 5 examines the applicability of PAMS to major modeling problems of a real world nature. This is accomplished by applying PAMS to the *IFIP Working Conference Problem*.

Finally, Chapter 6 summarizes the problems, contributions, and future research directions suggested by this work.

CHAPTER 2

Relationship of Database Activity to Other Approaches

In order to develop a model of database activity without "reinventing the wheel," one must have a thorough understanding of past efforts. The approach here is to highlight major modeling efforts of recent years, summarize the current state of such modeling efforts, and from this define the notion of *database activity*. Attempting to compare modeling efforts, however, can be difficult without a unified set of terminology and measurement criteria. Thus, some digressions must be made to lay groundwork.

2.1. The Premise of Data Modeling

Even though Codd's work on the Relational Model is generally perceived as the original work in data modeling, the concept has much deeper origins (see [Elliott1965]). All work in the field of database management revolves around the fact that "regularities" in any mass of information may be factored out and used to structure the remaining data. Early systems were predicated on an *ad-hoc* observation of what constituted "regularity," while later systems have been based on theoretical methods (with the Relational Model being the first accepted member of this later class). All of these are only a means to an end, however. As C. J. Date has noted [Date1983, pp. 182-183]:

> The primary purpose of any data model, relational or otherwise, is of course to provide a formal means of representing information and a formal means of manipulating such a representation. For a particular model to be useful for a particular application, there must clearly exist some simple correspondence between the components of that model and the elements of that application; that is, the process of mapping elements of the application into constructs of the model must be reasonably straightforward. To put it another way, a data model should possess some generally accepted *interpretation*.

If data models are to be compared, it is the author's contention that the primary factor must be ***the ease of understanding resultant databases***. This does not equate to how easily the data model can be understood, but to the combination of data model and data as perceived by whoever is trying to understand them. Since (at this point in history) *people* must understand databases, and since human understanding often works by analogy to past experiences, it is quite reasonable that data model ***interpretations*** based on "real world" structures should be central to "good" data models.

Unfortunately, the phrase ***data model*** still has no precise, agreed-upon meaning, even though the term is widely used. Consider the following two definitions expressed at the June 1980 Workshop on Data Abstraction, Databases, and Conceptual Modeling. Codd defines ***data model*** as follows [Codd1980, p. 112; emphasis added]:

> It is a combination of three components:
> (1) a collection of *data structure types* (the building blocks of any database that conforms to the model);
> (2) a collection of *operators* or inferencing rules, which can be applied to any valid instances of the data types listed in (1), to retrieve or derive data from any parts of those structures in any combinations desired;

> (3) a collection of *general integrity rules,* which implicitly or explicitly define the set of consistent database states or changes of state or both — these rules may sometimes be expressed as insert-update-delete rules.

Alternatively, McLeod and Smith take a different approach to define *database model* (which is used synonymously with *data model*) [McLeod1980, p. 20]:

> Specifically, a database model consists of four logical components:
> (1) a *data space,* which consists of a set of atomic *elements,* and certain *relationships* among them,
> (2) *type definition constraints,* which specify restrictions on the relationships in the data space,
> (3) *manipulation operations,* which allow elements to be created and destroyed, and their relationships modified,
> (4) a *predicate language* which allows individual elements to be identified by their logical properties (and selected from the database).

There are several inconsistencies between these two definitions, both with respect to terminology as well as to content. The most noticeable distinction is the numerical difference between three and four basic components. This particular difference is quickly resolved by the fact that *operators* subsumes both *manipulation operations* and *predicate language.*

Another seemingly major distinction can be resolved by clarifying the use of the word *type.* The most fundamental distinction in data modeling is that between the type and the value of an object. The type defines the characteristics of the object and thus remains constant while the value is expected to change over time. Complication arises, however, because types are themselves often categorized by type. (For example, *integer* and *floating-point* are each a "type" and a "type of numeric type.") In Codd's definition of the relational model the object types are data structures such as relation, domain, and key [Date1983, p. 182]. On the other hand, McLeod and Smith would consider the object types to be specific *instances* of these data structures such as, for example, an employee relation [McLeod1980, pp. 20-21]. With this difference in abstraction level resolved, it is clear that *type definition constraints* and *general integrity rules* are essentially the same.

Thus one is left with Codd's data structure *types* versus McLeod and Smith's *data space.* Equating these two is incorrect. As just mentioned, Codd's types are limited to a predefined set of modeling objects (such as relation, domain, and key). The concept of data space, on the other hand, includes the instances of such objects (for example, an employee relation) as well as the instances of objects defined by such instances (such as employee Fred Jones). Thus the two are in fact disjoint concepts, neither of which are captured by the other definition of *data model.*

Therefore, it appears that two, if not more, distinct definitions for *data model* are used by the database community. This fact quite often causes ambiguity – which definition applies can depend on context. For instance, three different definitions of *relational model* are required to cover the following cases: *the relational model* usually implies Codd's definition, *a relational model of a company (in general)* usually implies McLeod and Smith's definition, and *the relational model of the XYZ company* still a third. In short, the notion of a *model* means different things to different researchers or in different contexts.

For the sake of this work, however, a choice of available definitions must be made to avoid ambiguity. Based on the above discussion, the following is chosen:

Definition: **Data Model**

A **data model** consists of at least four equally important components:
(1) the *objects* which represent "real world entities" being modeled,
(2) the *object types* by which the objects may be classified,
(3) the *operations* by which the objects may be manipulated, and
(4) the *general integrity rules* which govern the legality of transformations accomplished with the operators.

In conjunction with this definition, it is also important to introduce a related term which shall be used throughout the remainder of this work.

Definition: **Modeling System**

A **modeling system** is a set of explicit definitions which are consistent with the general objectives presented in the definition of any particular data model.

For example, it is possible to define any number of different relational modeling systems which each preserve Codd's notions of relations, attributes, and non-procedural access. (See, for example, Lamersdorf's discussion in [Lamersdorf1983]). For many research-oriented models, however, the interpretation is only a weak abstraction of a single system definition.

2.2. Central Issues for Modeling System Comparisons

Any examination of the literature on data models yields a vast collection of modeling systems, proposed alternatives, and practical extensions. One early comparison [Kerschberg1976] considered twenty-three different models and probably at least twice that many new models have been introduced since then. It would be both inappropriate as well as impractical to list the features of even a representative percentage of those models here. Indeed, the goal of this work is not to produce "just another model" which fits together old ideas in one more new way; rather the goal is more subtle.

At this stage of data model research, a new model is worthwhile only if it provides some demonstrably new insight, not just new features or new implementation means. Thus, the strategy of the next few sections is not to examine past efforts simply for the sake of such efforts, but rather to place those efforts within a well defined whole. Only after this whole is understood may deficiencies of the whole be examined and new solutions proposed.

As noted in the previous chapter, Codd has suggested that all models are actually "semantic" to some degree, but gives no guidelines for determining how "semantic" a given model is. The most basic issue in this regard is simply the breadth of information which is addressed by the modeling system.

Definition: **Scope**

The **scope** of a modeling system bounds what can be represented using the modeling system. Effectively, the scope is a definition (formal or otherwise) of what information can be modeled by the system.

In remarks at the June 1980 Workshop on Data Abstraction, Databases, and Conceptual Modeling, L. Peter Deutsch noted the following [Brodie1980b, p. 40, numbering added]:

> There are three different kinds of knowledge each of which can be modeled.
> (1) knowledge about the real world,
> (2) knowledge about our representation of the real world, and
> (3) knowledge about the operations of the system being used to manipulate these models of the real world.

As will be shown, it is the degree to which these different categories are explicitly captured within database objects that primarily determines the semantic level of any given modeling system.

Nearly as important, however, is the fact that all modeling systems can be characterized by the degree to which two characteristics are coupled — the ***concepts*** and the ***roles*** used to construct an application of the system.

Definition: **Concept**

Modeling **concepts** are the specific object types, operations, and general integrity rules incorporated in the definition of any given modeling system. For example, in the case of many relational modeling systems, the concepts include: relation, attribute, domain, and key object types; SELECT, PROJECT, and JOIN operations; as well as a "values of key attributes cannot be null" general integrity rule.

Definition: **Role**

The **role** associated with an object in the application (which may or may not correspond to a real world entity) defines the reason for including the object in the database. Some examples of roles include concrete entity, event, parent-child relationship, and view definition. Depending on the circumstances, it is possible for an object to have more than one role.

Thus concepts are aspects of a modeling system while roles are aspects of an application being modeled with the system. The ***coupling*** of concepts and roles in different modeling systems ranges from "tight coupling" to "loose coupling" in a way which is analogous to communication between coupled processors. As shall be shown, in a tight coupling there is little, if any, distinction between the role and the concept, while in a loose coupling the two are removed from each other by several levels of abstraction. Tightly coupled systems are typically less flexible because a higher degree of consistency must be maintained. Loosely coupled systems are typically more difficult to construct and maintain given a large number of varied couplings. As shall be discussed in the next few sections, modeling systems with tightly coupled roles and concepts typically carry a large amount of implicit information. While often making simple applications easy to construct, this hidden information limits the power of the system in more complex cases.

2.3. Characteristics of Contemporary Modeling Systems

The concern of this section is to gain an understanding of the principles on which contemporary database modeling systems are based. The thrust of this examination is to categorize these systems through an examination of their various scopes and concept/role coupling degrees. The order in which the models are presented is inspired by McLeod and Smith's similar discussion [McLeod1980] although several additional categories have been added. Essentially, the order is from "least semantic" to "most semantic" and also approximates the historical order in which the models were developed. As shown in Table 1, representative models have been chosen to characterize each category although numerous additional models could be cited in almost every case.

2.3.1. Hierarchical and Network Models

Database systems were developed quite some time before the notion of data modeling, yet all such systems were based (to some extent) on common modeling concepts. Typically by an after-the-fact abstraction process, models have been developed based on these

Representative Model	*Types of Knowledge*	*Coupling Degree*
Hierarchical and Network Models	Deutsch 1	tight
Entity-Relationship Model	Deutsch 1	moderate
Relational Model	Deutsch 1	loose
Semantic Data Model	Deutsch 1,2	moderate
Semantic Hierarchy Model	Deutsch 1,2	loose
Relational Model/Tasmania	Deutsch 1,2	loose
Information Systems Model	Deutsch 1,2,3	varies

Table 1. Various Approaches to Data Modeling in Contemporary Systems

traditional approaches to the problem of data management. The most widely known such models are the *Hierarchical Model* and the *Network Model* as abstracted from IBM's *Information Management System* (*IMS*) [McGee1977] and CODASYL's *Data Base Task Group Report* (*DBTG*) [CODASYL1971] respectively. The basic characteristic of each approach is the definition of distinct roles for database objects, such as: segments, parent-child relationships, and sibling relationships for the Hierarchical Model; and records, sets, and owner-member relationships for the Network Model.

The modeling concepts underlying these models are a direct consequence of these roles. The coupling is so tight that a clear demarcation line is established between what constitutes one class of database object and what constitutes another; once a database object is given a role in some application of the model, the modeling concept used to express the database object becomes absolutely fixed. Thus, a database object which is part of a hierarchical application can define a segment, a parent-child relationship, or some other appropriate role. However, it must be defined in terms of exactly one such modeling concept and can never be interpreted under any other modeling concept once it has been defined.

These models (and many others based in traditional data processing concepts) represent the most rigid and inflexible view of data modeling that will be examined in this chapter; they do not allow for changing interpretation of the objects which they model. Likewise, the scope of the modeling domain overlying such models is the most restrictive. Of the three kinds of knowledge referred to by Deutsch, only knowledge about the real world can be captured.

2.3.2. Entity-Relationship Model

Chen's *Entity-Relationship Model* [Chen1976], represents a class of models which use somewhat more flexible modeling concepts than those discussed above (though the scope of the model remains the same). Rather than specify concepts which are the direct result of roles for database objects, the concepts are based on (one or more) ***abstractions*** of roles.[1] The

[1] It is important to note that role is being used as defined earlier in this chapter. This usage is distinctly different from *role* as used by Chen.

abstractions used by these approaches generally divide the roles into two distinct classes: ***entities*** and ***relationships among entities***, though similar models include other categories as well. For each such category, these models provide a modeling concept to express database objects that fall into that particular category.

Because a given object can change roles if necessary, this decoupling of concepts and roles allows a degree of flexibility not found in the traditional approaches. For instance, a parent-child relationship between two entities in a hierarchical application must always be viewed as such. It is a well known problem, however, that in some cases or under certain conditions, this same relationship might be more properly considered a child-parent relationship (e.g., a *PART* has a *SUPPLIER* verses a *SUPPLIER* which provides a *PART*).[2] The Entity-Relationship Model (and many similar models) allows this change to occur since all relationships (regardless of their specific type) are modeled in exactly the same way. Thus, the tight coupling of modeling concepts and roles of database objects is broken, allowing an object to change roles over time or to have more than one role at the same time without having either to restructure the database or to maintain two different representations of the same piece of information.

However, these models do not go far enough in terms of flexibility [Chen1976, p. 10, n. 1]:

> It is possible that some people may view something (e.g., marriage) as an entity while other people may view it as a relationship. We think that this is a decision which has to be made by the enterprise administrator [Steel1975]. He should define what are entities and what are relationships so that the distinction is suitable for his environment.

Hence, while these approaches do allow the role of a database object to change, this change is limited in scope to roles of a given class because different modeling concepts are used for different classes and no implicit conversion between such concepts is defined. (When conversion is necessary, it must be done by a user or program which is outside the scope of the model.)

2.3.3. Relational Model

The primary distinction between relational modeling systems and those just discussed is simply that relational systems do not couple roles and concepts. Rather than categorize objects into distinct classes, the Relational Model uses only one set of concepts to capture all database objects. This allows the same database object to be viewed as either an entity or a relationship. For example, the same modeling concept (in this case, a tuple) could be used to express (1) an entity, such as a *CUSTOMER*, or (2) a parent-child relationship between that entity and another entity, such as a *CUSTOMER AND THE CUSTOMER'S ADDRESS*.

In one very important sense then, the Relational Model is an ultimate data model. All database objects in any given application are represented in a uniform way, regardless of their roles. On the other hand, as noted by Schmid and Swenson [Schmid1975], the Relational Model has one major problem. As the size of an application based on the relational modeling concepts grows from demonstration to realistic levels, so does the potential for ambiguity when interpreting the role of individual database objects (particularly ***tuples*** and ***attribute values within tuples***). Thus, one does not know if a given database object represents

[2] As a typographical convention, names of object types (such as particular attributes, relations, or co-sets) shall be shown in *UPPER CASE ITALICS* throughout this work.

an entity, some particular kind of relationship, or possibly both at the same time.

One reason for this ambiguity is that any given application of a relational modeling system usually captures *less* information about the world being modeled than a corresponding application of a hierarchical, network, or entity-relationship modeling system. In generalizing the ability to represent roles, the Relational Model loses implicit information about roles. For instance, in a system where parent-child relationships are an inherent concept, the application includes the implicit fact that a child cannot exist without its parent. With a system that does not have a parent-child concept, such facts must be explicitly (rather than implicitly) recorded. While the entity-relationship model allows facts which are common to an entire class to be carried implicitly, the Relational Model includes almost no such implicit information. To explicitly include such information in relational systems requires it be represented by additional database objects (and most database designers fail to include these objects when they design an application).

2.3.4. Semantic Data Model

"Role description objects" tend not to be included in applications of the Relational Model simply because they actually fall in Deutsch's second category of modeling knowledge, that *about the representation* of the real world (rather than knowledge *directly about* the real world). Roles are an artifact of the representation of world objects as database objects. For example, there is no concrete entity in the real world which directly corresponds to "parent-child relationship." It is an abstraction of many individual relationships between parent objects and child objects, and each such relationship may be slightly different. It is the formalization of these relationships (as database objects or otherwise) that creates a need for the "parent-child relationship" abstraction.

While it is certainly true that a database description is nothing but knowledge of the representation of the world as the database, the database itself does not contain information about the representation. Thus, an application of the Relational Model (or any of the other models discussed earlier) defines only database objects, not other *application objects* such as role definitions. These models were never intended to capture such information as part of the application since it is outside the scope of their modeling domains. This realization has led numerous researchers to investigate extensions to the Relational Model (and to a far less extent, to the other models discussed thus far) which not only describe the data in the database but also the *semantics* of the data as well.

Perhaps the best representative of such an approach is Hammer and McLeod's *Semantic Data Model* (*SDM*) [Hammer1978, Hammer1981]. It provides a single concept, a *class*, to describe all database objects in much the same way as the Relational Model uses the concept of a relation. In addition to this, however, other SDM application objects use the concept of an *inter-class connection* to define alternate roles for the database objects defined as a class.

Unfortunately, the Semantic Data Model uses two types of classes: *base classes* to define traditional database objects and *non-base classes* for those application objects defined using the inter-class connection. On an entirely different plane of abstraction, this is the old problem of entities and relationships tied to the modeling concepts used to define them. While it seems natural to divide descriptions of database objects (i.e., base class definitions) from descriptions of roles for those database objects (i.e., non-base class definitions), the separation is of importance only in defining the object, not when refering to the object. For instance, when using a list of *EMPLOYEES IN THE TOY DEPARTMENT* and a list of

EMPLOYEES IN BUILDING 27 it is not important if one is derived from the other (or both from still a third class).

2.3.5. Semantic Hierarchy Model

Smith and Smith's work in connection with *Aggregation* and *Generalization* [Smith1977a, Smith1977b], later named the *Semantic Hierarchy Model* [Smith1980], takes the next, somewhat subtle step. Aggregation and generalization are similar to the inter-class connections of the Semantic Data Model in that each is a concept to specify that an application object is defined by a relationship among other application objects. One key difference makes the Semantic Hierarchy Model stand out, however [Smith1977b, p. 105, abstract]:

> Two kinds of abstraction that are fundamentally important in database design and usage are defined (by this work). Aggregation is an abstraction which turns a relationship between objects into an aggregate object. Generalization is an abstraction which turns a class of objects into a generic object. It is suggested that all objects (individual, aggregate, generic) should be given uniform treatment in models of the real world. A new data type, called generic, is developed as a primitive for defining such models.

Thus, Smith and Smith use a single modeling concept to express three distinct types of application objects in much the same way that the Relational Model itself uses a single concept to capture both entities and relationships. The Semantic Hierarchy Model represents a first step toward the integration of database semantics.

From a historical perspective it is worth noting, however, that the more unified Semantic Hierarchy Model preceded the more comprehensive Semantic Data Model in development. Unfortunately, while several important types of application objects can only be described using the Semantic Data Model, some degree of generality was lost in adding this further functionality. Thus these two models share much the same relationship as do the Entity-Relationship and Relational Models.

2.3.6. Relational Model/Tasmania

The next step in data model unification seems almost too obvious. If all database objects can be represented in a uniform way (as in the Relational Model) and if all application objects can be represented in a uniform way (as in the Semantic Hierarchy Model), one should at least examine the possibility of representing all such objects in the same uniform way. In other words, one should look for a single set of modeling concepts which can represent both the data and its description.

At first this may sound sacrilegious since the basic premise of data modeling has always been to extract the regularities from the data, using this as a (separate) description of the data. Upon further consideration, however, it is nothing more than the old problem of values, their types, and the use of types as values of still more encompassing types. The physical separation of these regularities from the data is only an artifact of the modeling concepts and database languages which have been used. Unifying the two into a single representation has many of the same benefits as the unification of database objects and the unification of application objects, only on a third plane of abstraction. For instance, it is consistent for the attributes of an *EMPLOYEE* relation to be described by tuples found in yet another relation, say *EMPLOYEE ATTRIBUTES*, rather than as text in some separate schema. Then one may use the same database language constructs to find out about the *EMPLOYEE* relation and about the *EMPLOYEE*s themselves. Indeed this blending of world knowledge and representational knowledge in a unification of schema plus data has been

cited as a key factor in the modeling of database semantics [McLeod1980]. Several modeling systems have taken this approach to heart. Most notable are *Query By Example* [Zloof1975] and *Relational Model/Tasmania* [Codd1979].

Query By Example is quite notable for the completeness of its application and database unification: the only way to create new relations is to enter data into a relation which defines the new one. However, its relevance here is limited since its semantic content most closely resembles the Relational Model rather than the semantic models which have just been discussed.

Relational Model/Tasmania (RM/T) unifies a full-fledged semantic application description system (on the order of that provided by the Semantic Data Model) with a complete relational modeling system. RM/T is not without its limitations, however. Unlike the Semantic Hierarchy Model, it does not use a single concept to describe all objects. While a single concept, the relational tuple, is used syntactically to capture all application and database objects, the semantics associated with such tuples are not as unified. Codd's three types of RM/T *entities* (associative, kernel, and characteristic) categorize these tuples with the same resulting complications as the different types of *classes* in the Semantic Data Model. Nonetheless, RM/T is the most completely unified model within this category, and thus has been chosen as representative.

2.3.7. Information Systems Model

In parallel with the trend toward the integration of semantics, attempts have been made to deal with the problem of database transactions. The basic idea is to incorporate Deutsch's third category of knowledge: that about *operations* on the system representing the world being modeled. The goal is to capture behavioral information (sometimes called "process definitions" or "system dynamics") rather than the static structural relationships which are central to the approaches discussed thus far.

A representative example of this approach is the *Informations Systems Model* [Foucaut1978, Rolland1979], though the modeling systems incorporating behavioral information are particularly diverse. The approach of such models is generally to describe the *object* (or *data*) being acted upon and the *operation* (or *process*) carrying out the action, then capture both within a *system model*. While each such model has a broader scope than traditionally associated with a data model, each is consistent with the data model definition used in this work. Additionally, the binding between the data and the process is also captured, with the resulting artifact often referred to as an *event*. Thus the modeling concepts underlying information system models always have at least three components: the data, the processes, and the bindings.

The basic problem with these systems is simply that they deliberately separate the operation causing the activity from the object being acted upon. Typically, operations are described using abstract algebras [Ehrig1978, Lockemann1979, Paolini1981], pre- and post-conditions [Leveson1980, Leveson1983, Mylopoulos1980, Weber1978], or some form of petri nets [Antonellis1981, Leonard1981]. Objects, on the other hand, are typically described with a modeling system based in one of the categories discussed in the last few sections. These models treat *objects* and *operations* in much the same way that the Entity-Relationship Model treats *entities* and *relationships* or the Semantic Data Model treats *application objects* and *database objects*. Even the arguments sound quite familiar [Brodie1980a, p. 59]:

> There's a tradeoff between modeling a fact as either a data object such as home-purchase

> or as an operation. They [*sic*] guy who records the fact "home is being purchased" may want to say "I want to invoke an operation," and give a couple of parameters. Someone else can say "what are all the instances of home-purchase that happened over the year?" The tax board may want to view them as entities. There's a degree of relativism involving operations.

In summary, once an operation is described, its definition becomes a fixed parameter of the application. Further, since operations are represented ***differently*** from objects, one cannot describe an operation which takes another operation as its object. While Information Systems style models do have a wide scope, they do not use a single modeling concept to represent all objects (including operations) as does the Semantic Hierarchy Model, nor do they integrate their representation and data as does the Relational Model/Tasmania.

2.4. Trends In Data Model Research

The thrust of the research reported herein is a desire to carry the semantic abstraction process one step further by reducing the large number of potential modeling concepts developed in previous work. Several trends point out the proper direction for accomplishing such a goal.

2.4.1. Static Templates

The traditional view of a database is that it is a collection of data about some portion of the real world. At any given instant in time, the current contents of the database is perceived as a ***snapshot*** of the world being modeled. Typically, the application or schema for such a database acts as what might be called a ***static template*** by which database objects are built and inter-related.

This analogy to photographic snapshots is apt. Snapshots (by definition) provide only a partial record of the real world from which they are derived because they are two dimensional while the world is three dimensional. In mapping from one representation to another, some information is lost and the remaining information may be subject to ambiguous interpretation. Unfortunately, such is also true of databases defined using static templates.

One can characterize the goal of data model research (and particularly, the goal of *semantic* data model research) as the process of bringing those snapshots closer to reality. However, the diversity of different research efforts has produced results in varied directions:

(1) Early work on data modeling can best be equated to that of the photographic pioneers. Learning how to take snapshots was no small accomplishment; different technologies produced different renditions of reality such as tintypes, brownie snapshots, and instant prints. The analogies for data models are the Hierarchical, Network, and Entity-Relationship Models .

(2) Reproduction of color added a new dimension to photography and, in data model research, so did the unification of representation concepts in the Relational Model.

(3) Models primarily concerned with the structural aspects of data (such as the Semantic Hierarchy Model) made sure the snapshot really is one of reality and not the result of darkroom trickery. Thus, the modeling of structure is equivalent to modeling the valid contents of a snapshot.

(4) It took the work of numerous researchers to change "snapshots" into "movies" and then to add "sound." These more comprehensive models of reality include the Semantic

Data Model and Relational Model/Tasmania.

(5) Those researchers concerned with behavioral modeling (i.e., those working on models such as the Information Systems Model) provide the same legality function for movies as do the structuralists for snapshots. In other words, modeling behavior is equivalent to modeling the legal transitions from snapshot to snapshot.

Thus, one can summarize most data modeling efforts with the following definition:

Definition: **Static Template**

A **static template** is an application which defines a database describing the *state* and possibly the *change of state* in the world being modeled.

2.4.2. Dynamic Templates

None of the data management research results described in the previous section is what might be termed a *dynamic template*, however. While a simple set of application objects may well describe the static state of a set of world objects (i.e., a snapshot) and a more complex set may describe a static progression of states for the world objects (i.e., a movie), no set of application objects has yet to be described which specifies how a progression may change dynamically.

Definition: **Dynamic Template**

A **dynamic template** is a static template extended to account for apparent change in the database due to differences in the perspective of the viewer or in the overall state of the database.

This requires some further explanation. As this is being written, the latest craze in arcade games is the application of laser disk technology. In such a game, the player appears to be watching a movie; but the player's participation (through the game controls) causes the sequence of events in the movie to change. In effect, each player sees his/her own different movie. In reality however, only a limited number of movie scenes are recorded on the laser disk and their sequence of play is dynamically constructed/modified based on the actions of the player.

Thus the basic principle of *dynamic templates* can be summarized as follows: while the contents of the database is finite and the number of possible views of that database is finite, the views need not all be predefined. Instead of constructing a (static) template for each view, a new view is constructed on demand by dynamically filling the parameters of the view definition. Moreover, unlike a laser disk game, as new database or application objects are defined, the entire application and all relevant views are adjusted accordingly.

Relational Model/Tasmania comes closer to a comprehensive model based on the concept of dynamic templates than any other semantic data model. If carried to a logical conclusion, its unification of application and database might be the equivalent of a movie about making movies. Add a few mirrors and the result could be a unified picture of a single movie describing how it was made, together with the context of the world around the process.

The remaining problem is how to reduce the complexity of RM/T while extending its functionality in the direction of the Information Systems style models. The key to the solution is a dynamic template modeling concept which accepts the fact that two snapshots of the same reality may look very different depending on from where they were shot.

2.4.3. Database Activity Revisited

Little has yet been said about the theme of this dissertation *per se*: the notion of *database activity*. This chapter opened with a quotation from C. J. Date concerning the need for an understandable interpretation behind every usable model. Database activity is meant to be such an interpretation. Unlike most contemporary models, the interpretation is not just a data structure such as a tree or table, but rather a structure and its variations:

Definition: **Database Activity**

> **Database activity** represents the state of the world being modeled and apparent changes in that state caused by a "rotation" of the database in one or more degrees of freedom as defined by the application.

Of particular importance are the ideas of "apparent change" and "degrees of freedom."

Apparent change is what distinguishes the position of database activity on the continuum of "least" to "most" semantic scope. Early models generally labeled as "syntactic" and based on static templates represent only the state of the world being modeled. Later models generally labeled as "semantic" but still based on static templates added the ability to represent changes in that state. Models of database activity capture additional semantics using dynamic templates which allow the definition of the representation *apparently* to change, while in reality, the definition is selected dynamically from some well defined set. In contrast as is discussed in Chapter 6, additional semantics could be captured by models which actually create new definitions for the representation.

Degrees of freedom indicates the fact that more than one factor may effect a person's view of reality. For example, the passage of time clearly changes one's view of most world objects. So does a change in the viewer's location, or a change of who is the viewer. Chapter 4 addresses these semantic dependency issues in detail.

The remainder of this work continues to pursue the notion of database activity as a useful interpretation and develops a *Prototype Activity Modeling System* (*PAMS*). Chapter 3 develops the basic, unified modeling concepts for representing the dynamic templates on which PAMS is based while Chapter 4 presents PAMS. Finally, Chapter 5 applies PAMS to a complex, large-scale example based on the IFIP Working Conference Problem and Chapter 6 summarizes the results of this work.

CHAPTER 3

Unified Modeling Concepts

In order to construct a model of database activity which is consistent with the principles of the previous chapter, one must have a single consistent set of modeling concepts which can be used to express all database and application objects. This must be true regardless of the roles any individual object may play in the definition of dynamic templates. Thus, in this chapter the goal is to develop a generic set of such objects independent of the fact that they will be used to model database activity — that task will be expanded in Chapter 4. The primary resulting concept, known as a *bundle,* encompasses the common concept of a *relation* as a special case.

3.1. The Premature Generalization Trap

People seem to build expectations based on the *majority* rather than the totality of cases involved in a decision and thus ignore special situations if they represent only a relatively small percentage of cases. In other words, people tend to generalize in a way that "forgets" the exceptions unless something triggers thoughts to the contrary: yes, birds fly; but no, ostriches don't. This *premature generalization trap*[1] often causes an application designer to make assumptions about the structure of data which later prove to be false. Once these decisions are reflected within an application, however, they prove difficult to retract or modify. One could argue that proper application design should avoid such problems. Unfortunately, (1) design time decisions may prove to be incorrect or, (2) the basis for such decisions may evolve over time in a way which invalidates the decision. The net effect of premature generalization is a factoring out of false regularities into the application design.

It is far beyond the scope of this work to consider generalized solutions to the premature generalization trap since it would be a premature generalization to do so. However, in the previous chapter it was pointed out that some models are more immune to this problem than others. In particular, it has been noted that the relational model entirely avoids the most classic example of the trap: the *non-entity relationship trap.* The relational model does not force a premature generalization of application objects into those which are *entities* and those which are only *relationships* since it uses only a single modeling concept to capture both. Additionally, Relational Model/Tasmania avoids the similar trap which forces other models to distinguish between application and database objects. RM/T instead uses a single database to capture and integrate both.

Two observations of the previous chapter are also quite relevant:

(1) RM/T comes closer to the expression of dynamic templates (and thus to a model of database activity) than any other model, and

---

[1] This term has been chosen in the spirit of Codd's *premature binary decomposition trap* [Codd1979].

(2) RM/T uses the relation as its basic modeling concept for both database and application objects.

It would be nice, therefore, to end this chapter immediately by saying that the Prototype Activity Modeling System (PAMS) developed in the next chapter also uses relations as its basic modeling concept. However, such is not the case. The use of relations as modeling mechanisms allows the database designer potentially to become the victim of several other instances of the premature generalization trap, which, it is the author's contention, are quite prominent in the complexity of RM/T.

(1) The *single value trap* forces a distinction between database objects which have only a single instance at any point in time and those which may have multiple instances.

(2) The *atomic value trap* similarly forces a distinction between database objects which are atomic in structure and those which are derived from several component database objects.

(3) The *named reference trap* forces all database objects to be referenced within an explicit naming structure rather than by their context among other database objects.

Each of these traps poses a particular problem when constructing dynamic templates. Thus, the modeling concepts behind PAMS require that the traps be examined and resolved.

3.1.1. The Single Value Trap

The name of an employee, the author of a book, and the color of a car, for example, are each traditionally regarded as single valued characteristics of any particular employee, author, or car. When captured by relational modeling concepts, this means that any database object (such as an *EMPLOYEE'S NAME, ADDRESS,* or *BIRTH DATE*) which is represented by an attribute of a tuple may have only a single instance at any point in time. Other database objects, however, which are represented by a tuple (such as an *EMPLOYEE*) may have multiple instances concurrently since these tuples form a relation.

The problem with this dichotomy is that many seemingly single valued database objects such as those cited above often have more than one possible instance:

(1) It is often necessary to record current *and* former names for employees.

(2) A single book or journal articles will quite commonly have multiple authors.[2]

(3) While the color of one car may be red, another may be red and black, while a third may be red, black, and white.

All the models discussed in the previous chapter including the Relational Model and RM/T *assume* that a database object has a single value unless the database designer explicitly captures the multivalued nature of the object in the database description. The problem with such a process is that it occurs only at design time. Once committed to a single valued database object, changing the design to allow multiple instances is no simple task. The solution as proposed herein is that all database objects should be assumed as potentially multivalued.

[2] For a particularly extreme case, the reader is urged to check the authorship of [Chamberlin1976].

3.1.2. The Atomic Value Trap

The name may differ, but as noted by Su, Lam, and Lo, *atomic data elements* appear in virtually all modeling systems [Su1981, pp. 262-263]:

> In the framework of the semantic association model, the real world of an enterprise can be modeled by a number of associations of concepts. Two types of concepts can be distinguished, [*sic*] atomic concepts and nonatomic concepts. An atomic concept is a nondecomposable, observable physical object (e.g., a person, a store), an abstract thing (e.g., the color of an object), or an event (e.g., an employee working on a project) that exists in reality. The meaning of atomic concepts is assumed to be understood and need not be described or explained. Atomic concepts can be grouped together to form *nonatomic concepts.*

Hammer and McLeod [Hammer1981] refer to atomic data elements as *entities* which correspond to actual objects in the application environment, while others use terms such as *object, entity* or *concept.* In the relational model the atomic data elements are *attribute values* while in RM/T they have been further refined into *immediate property attribute values.* Regardless of the model, however, it is often unwise to distinguish between atomic and non-atomic database objects. Just as is the case with the single value trap, many seemingly atomic database objects are abstractions of still other such objects.

Consider the pseudo-classic part-supplier database from Codd [Codd1970] and Date [Date1981]; in particular, consider the *COLOR* of a *PART.* The value for a part's color is always chosen from some finite domain of part colors (say "red", "green", and "blue" for example). Su, Lam, and Lo express the general consensus when they state that these colors are atomic data elements. However, one might naturally ask whether "red" means "Tuscan red," "Cornell red," "fire engine red," or "strawberry red". The concept "red" is really an abstraction (in this case, a generalization) of these and other underlying "types of redness" which themselves are nonatomic: "Strawberry red" may refer to the color of fresh strawberries, frozen strawberries, or strawberry ice cream. Similar abstractions can be made for many other attributes in the part-supplier database such as a part's weight or name as well as a supplier's name or city.

It would appear that this abstraction process could be carried down ad infinitum through finer and finer levels of abstraction. Eventually, however, such semantic breakdown of atomic elements into smaller pieces loses all relevant connection to the world object being modeled. This is not to say that such detailed breakdown is unimportant or totally irrelevant, but rather only that the breakdown is irrelevant from a perspective at some particular level of abstraction. Thus, the types of redness may not be relevant with respect to parts but may be relevant from some other view of the application such as the purchasing department's interaction with a paint supplier.

The point is that the notion of atomic structure is relative to ones viewpoint: things seem atomic when they are not necessarily so. A model which imposes atomic structure is overly restrictive and the traditional assumption of an "atomic" data level and abstractions only *above* that level inevitably leads to problems in the specification of the application. Su, Lam, and Lo [Su1981] cite several examples of such problems in discussing semantic changes. Specifically, their Class I semantic change involves the conversion of an atomic object (in their example, *SKILL*) into an aggregation of new objects (namely, *SKILL-NAME, INST-NAME,* and one which they fail to name). This type of change would pose far less of a problem if the modeling system assumed that the object may be broken down into finer detail at

some future point in time. The system should assume that every level of information in the database results from the abstraction of information below it and may be abstracted into levels above it. Such goals are consistent with the notion of *design for change* [Parnas1979].

3.1.3. The Named Reference Trap

All previous discussion of particular database objects in this work has referred to those objects by names such as *COLOR* or *ADDRESS*. Such named reference is a central theme of the relational model though it comes in several flavors: relations and attributes are explicitly named by the application and individual tuples are implicitly named by attribute instances which act as keys. Creating and remembering names for all database objects can cause problems. Codd noted, in discussing the deficiencies of network models, that some objects do not have natural names [Codd1970, p. 382]:

> One important effect that the view adopted toward data had on the language used to retrieve it is the naming of data elements and sets . . . . With the usual network view, users will often be burdened with coining and using more relation names than are absolutely necessary since names are associated with paths (or path types) rather than with relations.

The relational model itself also suffers from an explosion of names, though to a far lesser degree than earlier models. This is, in part, because names (in the form of values for specific attributes) are a major means for expressing connectivity between related tuples. RM/T lessens the dependence somewhat by its use of surrogates but it does not do so in a unified way since surrogates apply only to tuples in different relations and cannot be used to relate tuples in the same relation.

Using names to identify individual objects is a form of *absolute reference*. People, on the other hand, are also quite capable of referring to objects on a relative basis, by the context within which the object resides. Examples of this might be an employee's current name or name at birth and a car's predominant color. Clearly both types of reference are used in everyday conversation and some objects lend themselves better to one form of reference than the other.

Bolour and Dekeyser point out the distinction quite nicely in their discussion of absolute and relative reference to time values [Bolour1983, p. 43]:

> The view of time as a linear sequence of times points will be referred to as *absolute time*. This is the view that is adopted in most computerized information systems, where timing information is represented by calendar and clock events . . . . In this view, then, dates and clock times are convenient names for absolute time points. The use of absolute time is prevalent in scientific and administrative applications, where precise timing information is available for each recorded fact. But in most natural language historical accounts timing information is also conveyed by timing relationships such as "before", and "after". The view of time as a set of such relationships between events will be referred to as *relative time*.

After pointing out that this distinction between viewpoints of time is rooted in long standing philosophical controversy, Bolour and Dekeyser also observe that this absolute/relative dichotomy may also be applied to physical world concepts such as temperature, length, and weight. Supporting relative reference can substantially reduce the need for names, especially in cases where no natural name exists for a data item as is often the case in semantic data models.

3.2. Fundamental Principles

To avoid the traps just discussed while still preserving the basic benefits of the relational model requires a carefully defined extension of the relational model's basic concepts. The extension, to be defined shortly, is based on the controlled use of three fundamental principles: two forms of aggregation and one form of relative reference. These will be discussed in turn along with methods for combining their benefits.

3.2.1. Fundamental Aggregations

As has been discussed in the previous chapter, the term *aggregation* is most commonly attributed to Smith and Smith [Smith1977a, p. 406]:

> An aggregation is an abstraction which allows a relationship between named objects to be thought of as a (higher-level) named object. For example, a certain relationship between a person, a hotel, a room, and a date may be abstracted as the aggregate object "reservation". The name "reservation" may be used without bringing to mind all the details of the underlying relationship.

Other authors have since noted the fact that aggregation itself comes in many forms. The particular form Smith and Smith describe has come to be called *Cartesian aggregation* since it represents a subsetting operation over the Cartesian product of the domains which define the objects involved in the relationship. Cartesian aggregation is the primary solution to the atomic value trap since it is a fundamental method for giving structure to a value.

A second and equally important form of aggregation is mentioned by Smith and Smith and discussed extensively in [Codd1979] and [Hammer1978]. *Cover aggregation*, as it has since come to be called, involves the arbitrary collection of a (potentially large) number of instances into something that can be regarded as a higher-level instance. The two most commonly cited examples are the grouping of ships into convoys and the grouping of people into clubs.

The distinction between cover aggregation and Cartesian aggregation often seems superficial since it could easily be argued that individual ships are "attributes" of a convoy and individual people are "attributes" of a club. The true distinction might best be illustrated thus: removing any or all ships from a convoy still leaves one (at least nominally) with a convoy while, on the other hand, removing any portion of a reservation (such as the person or the date) invalidates the entire reservation relationship.[3] Expressed in a more general fashion, the distinction here is between aggregations which do or do not depend on the roles of their constituent database objects; Cartesian aggregation is role based while cover aggregation is not.

Definition: **Cartesian Aggregation**

A database object which is defined by the application to be a **Cartesian aggregation** represents a collection of roles for other database object instances. Each such role must be filled by an instance in order for the aggregate instance to be considered meaningful.

---

[3] One might argue that it is possible to have a reservation without values for one or more of these attributes since null values may be used in place of the missing data. Clearly, however, such a reservation cannot be used in the same ways as the reservation postulated above simply because, by definition, it is incomplete. In a sense, such an aggregation represents some lesser type of "reservation" relationship.

Definition: **Cover Aggregation**
A database object which is defined by the application to be a **cover aggregation** represents a collection of database object instances.

Thus, each slot in a Cartesian relationship must be filled and, further, must be filled with the correct type of object, if the relationship is to hold. Cover aggregation, on the other hand, requires only arbitrary grouping without specific roles. Further, the arbitrary nature of cover aggregation also allows the number of slots (in this case the size of the convoy) to vary. For this reason cover aggregation acts as the fundamental principle for avoiding the single value trap since any object defined by a cover aggregation may naturally contain multiple instances.

Of basic interest in the construction of unified modeling concepts is the notion that all relations are a pairing of a cover with a Cartesian aggregation. Smith and Smith sensed this point in their original discussion of aggregation, though they did not pursue it in depth. One of their example relations (expressed in a PASCAL variant) which contains university course information [Smith1977a, p. 407] is shown below. In their terms, "aggregate" denotes a Cartesian aggregation while "collection" denotes a cover aggregation. Any relation may be similarly expressed.

```
type course = aggregate [C#]
                C#: number;
                CH: credit-hours;
                D:  description
              end;
var courses: collection of course.
```

When used in this fashion, it is important to note that the members of the cover aggregation are ***homogeneous*** because they are each defined by the same Cartesian aggregation. All cover aggregations to be used in this work are homogeneous, though the basic definition of cover aggregation (as well as accepted usage) does not require this additional constraint.

In order to further increase their usefulness, it is sometimes necessary to place specific limits on the cardinality of a cover aggregation. Such ***cardinality constraints*** might limit the number of birth date instances associated with an object to only a single value or the number of students associated with a university class to the number of seats in the classroom. While such constraints are not commonly found in relational modeling systems, they are not new. See, for example, [Rolland1979].

3.2.2. Positional Reference

One method for achieving relative reference in a database is to rely on the position of an instance relative to some ordering of all related instances. For example, one can refer to particular employees relative to an ordering of their hire dates. This ***positional reference*** methodology acts with respect to cover aggregation in the same way as does the indexing of arrays, provided there is some ordering function on the elements of the cover aggregation. In this way then, the concept of an array acts as a suitable interpretation for cover aggregations supporting positional reference; the elements of either are each homogeneous and often constrained to a certain total cardinality.

Using positional reference in connection with both cover and Cartesian aggregation is tantamount to supporting ordered relations. One might argue that this is a sacrilege as far as Codd's original intentions are concerned since dependencies on ordering, indexing, and

access paths were the three principal forms of data dependencies eliminated by the original Relational Model. However, one must distinguish between *presentation order* and *storage order*. It is the latter which Codd sought to eliminate. The former was eliminated from the model only as a by-product of the latter. Several researchers have suggested that presentation ordering be supported in at least a limited fashion, as evidenced by event precedence (and thus successor graphs) in RM/T [Codd1979] as well as the ordering primitive for member derivation in SDM [Hammer1981].

In choosing to support positional reference, a decision must be made about its relationship to named reference. One can support each as separate fundamental principles, or alternatively, model one in terms of the other. Most current relational systems support positional reference only by encoding the ordering scheme as key values within the relation. As far as the application is concerned, these data values then act as names for their associated tuples. A few relational systems do support presentation ordering directly by allowing the specification of "sorting order," but this is usually supported only for retrieval of data and not integrated with the remainder of the model.

Bolour and Dekeyser's modeling of relative time takes the opposite approach in that all references to absolute time are modeled in terms of relative references [Bolour1983]. This is the view adopted here for handling named references. Naming schemes are defined on top of a positional reference system. This is done because it is particularly simple to model named reference in terms of positional reference; merely list the names of the objects in one cover aggregation and use this list of names (now treated as data) as a positional reference to the identically ordered list of original data objects. The same cannot be said if one tries to model positional reference in terms of named reference. The result is an explosion of artificial names, especially when one tries to order the same set of data with multiple ordering functions.

3.3. Definition of a Bundle

With the preliminaries aside, it is now possible to define the modeling concept central to this work, the *bundle*, as a structured combination of Cartesian aggregation, cover aggregation, and positional reference. This informal choice of term comes by analogy: whenever one has a bundle of something, it can be manipulated either as a whole or it can be unbundled into its constituent parts which are each instances of some (particular) kind of object. Further, when a bundle is unbundled, it is usually done so one object at a time, and thus in some reproducible order . . . unless, of course, the bundle was dropped on the floor. At different times or under different conditions, the bundle may be undone differently and thus the objects in the bundle may have several possible orderings. Once the bundle is undone, individual items may be selected and grouped to form new bundles, if desired. In some cases, each item in the bundle may be identified by its name, but such is not an absolute necessity.

Definition: **Bundle**

A **bundle** of objects is a homogeneous cover aggregation of Cartesian aggregations of objects. The cover aggregation may be ordered by one or more total ordering functions and may be constrained to any particular number of cardinalities. The objects contained in the Cartesian aggregation may be simple instances of data values or may be other bundles of objects.

The implication of this definition produces the last major leap beyond ordinary relations. That is, bundles may be nested to arbitrary depth, and (with suitable operations as

will be defined in the next few sections) this depth may change over time or even become transparent. Further, a significant degree of ***information hiding*** [Parnas1972] is thus supported.

3.3.1. Description and Notation for Simple Bundles

In order to give examples of bundles, it is convenient to use a number of notational conventions for bundles and their constituent parts. As in earlier chapters, simple data instances will be denoted by ordinary text as in

BOLT 1230 27 red

while names for objects will be shown in upper case italics.

PART NAME QUANTITY WEIGHT COLOR

A Cartesian aggregation will be shown following the convention often used elsewhere for tuples: a comma separated list surrounded by parentheses.

(Bolt,1230,27,red)
(*PART NAME,QUANTITY,WEIGHT,COLOR*)

In this case the top Cartesian aggregation would most likely represent a database object (since it contains instances) while the lower one would represent an application object since it contains object names. The two are *not* notationally distinguishable, however, since it is a goal of this work that one concept (and thus one notation) represent both. In this way, then, the appropriate interpretation of role can depend on usage.

In a similar fashion, cover aggregation will be represented by a comma separated list surrounded by curly braces.

{red, green, blue, brown, yellow, red, orange}
{*PART NAME,QUANTITY,WEIGHT,COLOR*}

It is important to realize that the two aggregations

(*PART NAME,QUANTITY,WEIGHT,COLOR*)
{*PART NAME,QUANTITY,WEIGHT,COLOR*}

are not the same. The former represents some well defined role-based relationship which cannot be added to nor deleted from without a change in semantics, while the latter is merely a list of names which may be freely modified in quantity.

Further, since all cover aggregations used in this work are homogeneous, it would be improper to find mixtures of objects such as

{bolt, *QUANTITY*, red, *COLOR*}

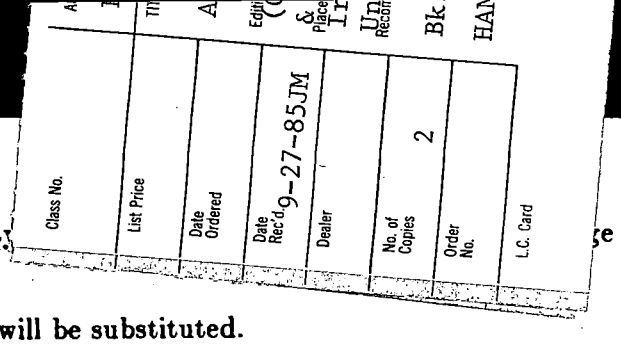
unless there were some type defined which included both application and database objects. Notationally, then, the equivalent of the Smith and Smith university course information relation could be shown as either of the following depending on the names one might choose for attributes.

{(*C#, CH, D*)}
{(*NUMBER,CREDIT-HOURS,DESCRIPTION*)}

As a convention for associating a name with a bundle, a labeling notation is used.

COURSE: {(number, credit hours, description)}

It should be noted, however, that naming is not required and, as will be explained in the next section, is accomplished through positional reference. Thus it is more accurate to think of this as a macro substitution — where *COURSE* is found in the context of a definition, the



corresponding description (after the colon) will be substituted.

This notation for a relation is not strictly accurate nor consistent with the notion of a bundle, however. Everything is fine if this is only regarded as a relation description with, *NUMBER*, *CREDIT HOURS*, and *DESCRIPTION* merely being names to be associated with data. However, if this notation is meant to represent the extended relation when data is substituted for names, the resulting groupings are incorrect.

INCORRECT: {({101,102},{4,4},{Intro,Advanced})}
CORRECT: {(101,4,Intro),(102,4,Advanced)}

Thus to produce the actual relation from its description requires the application of an operator rather than simple substitutions.

3.3.2. Generalizing the Description of Bundles

As a by-product of the above discussion, at least one representation for positional reference ordering functions has already been introduced — the comma separated list. One must be careful with such a notation, however, since it implies a storage order rather than a presentation order and since it is difficult to use when the order is defined by a function or when the aggregation is multiply ordered. Similar problems arise when one attempts to associate multiple names or multiple cardinality constraints with the bundle.

At this point, the arguments of this chapter take on a subtle hint of recursion. The problems just mentioned arise out of the various premature generalization traps discussed earlier. The notation usable for simple cases suffers by assumption about single level structures and single valued instances. Thus, an alternative view is worthy of consideration.

It takes five categories of information to specify any particular bundle completely:

the data values contained in the bundle (i.e., *extension* of the bundle),
the name(s) associated with the bundle as a whole,
the role identifier(s) associated with the Cartesian aggregation,
the ordering function(s) associated with the cover aggregation, and
the cardinality constraint(s) associated with the cover aggregation.

These items of information, taken together, clearly form a Cartesian aggregation of items with different roles. Further, each item individually is at least a cover aggregation of potentially many individual values.

Thus, it is appropriate to represent a bundle as a Cartesian aggregation:

(*DATA VALUES, NAMES, ROLES, ORDERS, CARDINALITIES*)

It has already been noted that *DATA VALUES* has an additional structure in the form of a cover aggregation of Cartesian aggregations. It should now be obvious that *NAMES, ROLES, ORDERS,* and *CARDINALITIES* are also multivalued structures and are, in particular, all at least cover aggregations. Thus, the same university course information relation of Smith and Smith might better be denoted:

({(101,4,Intro),(102,4,Advanced)}
{course}
{number,credit-hours,description}
{}
{})

At this point the course information does not fully exploit the notation since it is still constrained by the definition of a relation; i.e., the relation is not ordered, is not cardinally

constrained, and has only a single name. Similarly, all Codd relations could be described in this way.

Continuing the recursive argument, since names, role identifications, ordering functions, and cardinality constraints are all cover aggregations, one should support Cartesian aggregation, ordering functions, and cardinality constraints on these cover aggregations. Expressed somewhat differently, one should specify the *NAMES, ROLES, ORDERS,* and *CARDINALITIES* of a bundle by other bundles. As with all recursive arguments, however, this point is not easily taken and must, of course, eventually terminate. Thus, some additional discussion is in order.

For instance, consider a list of books in a library. Three common ordering functions for such a list would be *BY-AUTHOR, BY-TITLE,* and *BY-SUBJECT,* each as found in a card catalog. Such a catalog is often referred to as an *AUTHOR/TITLE/SUBJECT INDEX,* and thus one quite logical ordering for the list of ordering functions is precisely:

{BY-AUTHOR, BY-TITLE, BY-SUBJECT}

It may appear at this point however, that the argument has been recursively painted into the corner so to speak. Since there was *one* logical ordering of the ordering functions, there might just as well be *two* (or more) such orderings. Thus one has multiple values for the attribute of the data model that represents the ordering of the ordering functions and thus, quite logically, this too should be represented as a bundle on which there *may* also be multiple orderings . . . and so on, seemingly ***ad infinitum.***

Such a continued explosion of application description information is not *semantically* possible, even though it is syntactically. Given the author, subject, and title orderings for the original data, these ordering functions can be ordered in six distinct ways yet only a few of those orders would be of concern to anyone using such a database. While all of the orderings are indeed possible, only a few of them are semantically useful in the data modeling process; the other orders eliminate themselves from consideration in much the same way that meaningless tuples from a Cartesian product do not occur in the resulting relation. Thus, while the recursive consideration of orderings for ordering functions seems to explode combinatorily, the opposite is in fact the case. (For those still unconvinced, consider the impossibly numerous possible number of ordering functions for the library books themselves — only a *very* few of those orderings are meaningful enough to be captured by any application.) The number of orderings which need to be considered decreases with "distance" from the level of the original data itself, finally reaching some point where the ordering exhibits no further semantic connection to the model.

Similar examples can be given to show that *NAMES, ROLES,* and *CARDINALITIES* should also be specified as bundles rather than just cover aggregations. The importance of this result is that any data bundle may play the role of the name specification, role identification, ordering function, or cardinality constraint for another bundle. Thus, a single concept may be used to model both application and database objects, yet be immune to many premature generalizations about the nature of such objects.

3.4. Operations on Bundles

As was noted in Chapter 2, structural modeling concepts such as those just discussed are only one component of a data model. It is perhaps even more important to have robust operations defined over the structures so that data manipulation may be performed. The

operations defined over bundles in the context of this work consist mainly of common relational operators extended in a suitable fashion to work over bundles.

Since these operators have been discussed at length elsewhere [Codd1979], only a brief sampling is summarized here to clarify how the operators may be viewed in the context of bundles rather than relations.

* SELECTION produces a bundle where the value instances of the cover aggregations are a conditionally defined subset of those in the original bundle.
* PROJECTION produces a bundle where the value instances of the cover aggregation are each restructured according to a simpler Cartesian aggregation.
* JOIN produces a bundle where the value instances of the cover aggregation are a conditionally defined subset of the Cartesian product of the cover aggregations of two bundles.
* Statistical aggregation operators (such as SUM, COUNT, and MAX) reduce the members of the cover aggregation to a single value of some primitive, though not necessarily mathematical, data type.

Several operations are indirect extensions of other existing operators:

* BUNDLEUP takes an arbitrary number of objects and forms them into a bundle. It is assumed that the objects are homogeneous in type and thus that the description of the Cartesian aggregation may be taken from any of the objects being formed into the bundle. The bundle is given null bundles for the names, ordering, and cardinality components. This operation is most similar to Codd's SETREL.
* UNBUNDLE converts a bundle of bundles into a bundle of the nested bundles' data values. The order of the resulting bundle is achieved by composing the ordering functions of the outer bundle with that of the nested bundle. (The inner bundles must have the same function.) It most closely resembles a repeated UNION operator. The main difference is that bundles are not sets and, thus, no duplicate removal is performed.

For the many examples used later in this work, operations are expressed as nonprocedural queries using a syntax in the style of SQL (previously called SEQUEL [Chamberlin1974] or SEQUEL2 [Chamberlin1976]). Thus for example, to select the description of all courses numbered under 100, the following would be used:

SELECT *DESCRIPTION*
FROM *COURSE*
WHERE *NUMBER* < 100

Such syntax is, at best, a preliminary way of expressing complex operations on bundles and thus should be considered only as a vehicle for discussion. It may be more desirable to base an actual implementation of bundle operations on a more versatile language such as Wasserman's PLAIN [Wasserman1979, Wasserman1980].

Two important issues must also be kept in mind. The first is simply that full support for null values must be defined into the operators since bundles may incorporate null valued components. Second, and in some respects more important, is the fact that all operators must work transparently on individual data values as well as bundles. Thus implicit type conversion must occur between values and bundles when an operator/operand incompatibility would otherwise invalidate an expression. The reason for this is that queries must be

immune to changes between simple objects and objects which have additional structure and/or multiple instances.

This implicit conversion presents no particular problem when applying a bundle operator to a simple value since the value may be treated as a degenerate bundle. However, the opposite is the case when a scaler operator (such as plus) is applied to an entire bundle. To substitute a single value in place of what is known to be a bundle obviously requires some degree of data abstraction or even data loss. But sometimes it is this loss of semantic content that is desired from the perspective of the operation being attempted. While one can, of course, explicitly reduce the values by a statistical aggregation, there must be a default action which occurs when no such operation is specified. The operation used here shall be referred to as *nomination* of the "best" or "most representative value." Bundles support this *value nomination principle* quite nicely with the convention that the chosen value shall be the *first* value as determined by the ordering of the bundle. (If the bundle is multiply ordered, nomination is applied recursively to choose the best, i.e., *first*, ordering.)

3.5. Goals Revisited

The goal of this chapter is best summarized as the development of a basis on which the larger goal of a dynamic template modeling system can be achieved. The *bundle* and its associated operators achieve this by avoiding several *premature generalization traps* inherent in past approaches; yet this unified set of concepts benefits from past efforts because it is a logical extension of many relational modeling concepts. In essence, a bundle is a relation extended to support structural nesting, positional reference to tuples, and multiple instances of attribute values. Further, bundles may be used to describe other bundles. Thus, the concepts benefit from a unified representation of database and application objects in much the same way as do other advanced modeling approaches.

CHAPTER 4

The Prototype Activity Modeling System

This chapter presents a framework for expressing dynamic templates of database activity. Called the *Prototype Activity Modeling System* (*PAMS*) because of its preliminary nature, the framework is based on the view that all elements of a database description (i.e., application objects) are viewed as roles played by other objects. All relationships are dynamic in nature, including those which describe dynamic behavior and those which describe the role of an element in the description. To develop the modeling system, the roles objects may play are derived via an examination of the semantic dependencies which occur among related objects. From this, two guiding principles are derived: the ***non-procedural thunk principle*** and the ***activity monitoring principle***. Both principles add a significant degree of dynamics to otherwise static application templates and thus provide the key concepts necessary for at least one approach to modeling database activity.

4.1. Overview of the Prototype Activity Modeling System

The Prototype Activity Modeling System can be viewed as a progression of modeling systems each of which incorporate additional semantic information. For this reason, the approach taken is to present a series of levels, as shown in Table 2, which encompass succeedingly larger portions of the complete PAMS. Each of these levels could be used as a modeling system, but, as shall be seen, each also has one or more significant shortcomings. These limitations arise principally because each level adds one additional type of semantic abstraction. Thus, the Prototype Activity Modeling System is concerned with *four* distinct types of database semantics. Each can be expressed as a class of *dependencies* between one database object and the definition of another.

Definition: **Containment Dependency**

Containment dependencies include any constraints on the logical inclusion of one item within another. For example, a window is usually considered a component of some larger entity such as a wall.

Level:	*Basic Concept*	+	*Role*	=	*Next Concept*
0:	object types	+	structure	=	database description
1:	database description	+	data values	=	application
2:	application	+	scripts	=	tasks
3:	tasks	+	clocks	=	activities

Table 2. Modeling Levels within PAMS

Definition: **Feedback Dependency**

Feedback dependencies include any constraints which are due to particular data values in the application. For example, a student would have a grade for a particular class only if that student was enrolled in the class.

Definition: **Operational Dependency**

Operational dependencies include any constraints which are due to the way in which the data is being used. For example, the type of cargo carried by a particular ship would be part of the ship's definition only if that information was used within the application.

Definition: **State Dependency**

State dependencies include any constraints which are due to the fact that an event took place. For example, a building could be torn down only if that building was previously built.

No pretext is made by this work that these are the only types of dependencies which should or could be addressed within the context of dynamic templates. Rather, any complete model of database activity should be able to include at least dependencies of these types, and each such dependency should be dynamically subject to change (over time or due to changing requirements).

Throughout the discussion and development of the various levels, a single example will be used to illustrate these different forms of dependencies. Referred to as the Research Grant Example, it consists of a hypothetical collection of information about government supported research grants. Although not yet realistically possible, the collection is meant to contain a complete record of all such grants ever supported by the United States Government. Thus, the following notions will prove relevant over the course of this chapter:

* Queries against the database might range in scope from those about the total number of grants by a branch of government to those about the graduate work of particular principal investigators.
* Queries about the total number of active grants will yield different results at different times as the data in the database changes over time.
* A query about the status of a grant proposal under review might yield more than one result since the grant may be under review by more than one agency.
* The information available about the results of a grant review may vary from grant to grant since the grant review process may differ from agency to agency.

4.2. Containment Dependency

As discussed at length in Chapter 2, though the name may differ, the concept of *object types* is central to all database modeling systems, including PAMS. These systems do vary, however, in the types of *relationships among object types* which they support. Most semantic modeling systems incorporate one or more forms of aggregation which taken together allow an entire *structure* of relationships to be specified. Combining the object types with this structure, such as can be done in a unified way using the *nested bundle principle* introduced in Chapter 3, produces a *database description.*

It should come as no surprise that nested bundle descriptions constitute the fundamental basis for the Prototype Activity Modeling System. This initial level of the modeling

system encompasses no new concepts but merely establishes a baseline on which the rest of the model is built (hence the *zero* numbering in Table 2). Being very restrictive in nature when compared to other semantic data modeling systems, this level allows the specification of object types as either entities or bundles of entities — ***nothing else***! This level, then, captures hierarchical structure among entities, and because bundles are based on cover and Cartesian aggregations, this fundamental level of PAMS allows only the specification of containment hierarchies.

In the example, the hierarchical structure might recursively divide as follows:

U.S. Government
Branches (e.g., Legislative and Executive)
Departments (e.g., Defense and Commerce)
Granting Agencies (e.g., NSF, NIH, and DARPA)
Divisions (e.g., Computer Science, Math, and Health Science)
Sub-divisions (e.g., Theoretical Computer Science)
Grant Receiving Entities (e.g., University of California)
Locations (e.g., U.C. Irvine)
Departments
Principal Investigators
Academic Backgrounds

Each aggregation defined with PAMS consists of a bundle of names for the aggregated entities (in the same way that a relation is denoted as a collection of attribute names). Thus, using the notation introduced in the previous chapter, one possible (though incomplete) containment hierarchy representing the database might be:

US GOVERNMENT:
{(branch)}

BRANCH:
{(name, department)}

BRANCH.DEPARTMENT:
{(name, agency)}

AGENCY:
{(name, division)}

DIVISION:
{(name, sub-division)}

SUB-DIVISION:
{(name, institution)}

INSTITUTION:
{(name, location)}

LOCATION:
{(name, coordinator, department)}

LOCATION.DEPARTMENT:
{(name, size, address, telephone, grant)}

GRANT:
{(id, title, principal investigator, budget)}

PRINCIPAL INVESTIGATOR:
{(name, title, telephone, background)}

BACKGROUND:
{(professional, graduate, undergraduate)}

Several points about this database description are worthy of note. For one, it is not required that the Cartesian aggregation attribute names be unique across the database. For convenience, a "dot notation" (as in *BRANCH.DEPARTMENT*) is used to clarify apparently ambiguous references to names. However, this ambiguity is only an artifact of the notation being used. Recall from the previous chapter that the description of *DEPARTMENT* would be wholly contained (as a nested bundle) within the bundle of *BRANCH*, thus providing a true scoping of names. It is also possible to reference, say *INSTITUTION.GRANT*, as a shorthand for a more complex reference, such as *INSTITUTION.LOCATION.DEPARTMENT.GRANT*, because of the hierarchical structure contained in the definitions. The result of the reference would be no different than if *GRANT* were a direct attribute of *INSTITUTION*; bundle nesting provides a non-traditional form of data abstraction which hides ***intermediate*** levels of structure and detail. Further, many of the names used in the sample containment hierarchy are ***not*** mandatory since it will sometimes be more appropriate to reference data on a positional basis. For instance, one might use the first attribute of the Cartesian aggregation as an identification of the aggregate object rather than having attributes called *NAME* or *ID*. Such conventions form the principal mechanism for avoiding the named reference trap when using PAMS.

Another basic principle within this level of the modeling system is that object type hierarchies, such as the one presented above, are ***open-ended***, that is, without upper or lower bounds. In the Prototype Activity Modeling System ***every*** entity is considered to be an abstraction of lower level entities. In this way, PAMS avoids many of the problems associated with the atomic value trap. Also note that in order to avoid the single value trap the description does not reveal which attributes are multivalued and which are not. For example, given a *GRANT*, the *TITLE* attribute may or may not contain only a single value, and the *PRINCIPAL INVESTIGATOR* attribute may actually be a structure which describes co-investigators. In either such case, however, the ***value nomination principle*** produces only a single *TITLE* or the *NAME* of a single principal investigator if either attribute is used in a context which requires a single value. In these cases, the detailed information regarding multiple values as well as any additional sub-structure is hidden.

Obviously, the infinite hierarchy implied by such principles is impractical for implementation on a finite machine, and any particular ***implementation*** of the Research Grant Example will have upper and lower abstraction bounds. But in the Prototype Activity Modeling System these bounds may change over time or from one instance of the database to another. This is practical with PAMS because a single modeling concept, the principle of nested bundles, is used to capture both ***objects*** and ***object type descriptions*** (or in other words, both database objects and application objects). An object thus may take on a role as a new object type description, extending the bounds of the containment hierarchy in the process. Therefore the defined environment of the database becomes somewhat dynamic, rather than static; the database description remains ***unbounded*** in order not to subject the database to artificially and permanently imposed bounds.

4.3. Feedback Dependency

The next level of PAMS is concerned with the apparent distinction between the application and the database, or in other words, between the intention and the extension of the database. At this level, the *database description* of level zero is combined with the corresponding *data values* (the extension) to form the complete *application*. A fundamental premise of dynamic templates is that application and database objects share a common representation and therefore the definition of the intention may depend on the extension. Just as the limits of the containment hierarchy should not be artificially bound by the database description, neither should the internal structure of the hierarchy itself.

Consider those portions of the Research Grant Example which contain telephone numbers (of principal investigators, referees, agency contacts, etc.). While generally considered atomic in nature, previous arguments suggest that they can be defined as an abstraction of lower level entities. In the United States, for example, the definition is a Cartesian aggregation of *AREA CODE*, *EXCHANGE*, and *NUMBER*. In many countries, the aggregation is defined as *COUNTY CODE*, *ROUTING CODE*, and *NUMBER*, while for still other countries the *ROUTING CODE* is omitted. Thus the structure of a telephone number varies depending on the country of its origin; the *structure* of one object depends on the *value* of another.

Such dependence is not limited to aggregations of individual values. Any abstraction in the containment hierarchy may be conditionally selected by the value associated with some other object in the database. Thus, in the Research Grant Example it is possible to specify that some granting agencies (such as NSF) have a division/subdivision structure while other agencies consider that all their grants belong to a single pool, obliterating the need for multiple levels of structural hierarchy.

This notion of conditionally defined abstraction differs quite radically from the traditional approach to database construction. Rather than define a static database structure and then separately populate the structure with the extension of the database, conditional aggregation implies the existence of feedback loops. This does not say that the intention of a database cannot be defined *a priori*, but only that the intention of any particular object is not necessarily known until that particular entity is identified. Some semantic data modeling systems have included limited support for conditional abstraction. For instance, RM/T [Codd1979] uses graph relations to support alternative generalization. Such support is limited since different concepts are used to express application and database objects or because conditional abstraction cannot be added dynamically. On the other hand, there are systems such as Query by Example [Zloof1975] which do integrate both types of objects yet do not allow feedback loops.

The Prototype Activity Modeling System implements conditional abstraction in the following manner. At any point in the database structure where a single abstraction traditionally is specified, multiple abstractions may be conditionally specified instead, provided a connection is established to an object whose *value* delineates the choice of which abstraction is to be used. Based on the extended relational operators and SQL-style syntax introduced in Chapter 3, this connection may be established by replacing the aggregation description with a query specification which yields an appropriate description. For a simple case with just a few countries the telephone number problem can be solved thusly:

TELEPHONE:
SELECT *
FROM *TELEPHONE STRUCTURE BUNDLE*
VIA POSITION OF *ADDRESS.COUNTRY* IN *COUNTRY BUNDLE*

or populated with sample objects (to show nesting and avoid named references):

TELEPHONE:
SELECT *
FROM
{({area code, exchange, number}),
({country code, routing code, number}),
({country code, number})}
VIA POSITION OF *ADDRESS.COUNTRY*
IN {(US), (UK), (Panama)}

The more complicated case where multiple countries share the same aggregation is actually solved in the same way except that US, UK, and PANAMA are used as ***types of telephone numbers*** rather than as countries. Thus, *ADDRESS.COUNTRY* must first be converted into one of these types rather than used directly. This, however, involves another type of dependency and thus must await another level of the PAMS. (Alternatively, longer lists of countries as aggregation formats could simply be used; this, however, requires duplication of formats and is, therefore, less desirable.)

The principle of embedding executable database description derivations within other definitions is fundamentally important to PAMS. In further discussion, this embedding will be referred to as the ***non-procedural thunk principle***. This terminology follows quite naturally from the use of an SQL-style query specification as the definitional vehicle. A conventional thunk is a procedure which is passed in place of a literal value; this procedure is then transparently executed whenever the value is actually needed for computation [Ingerman1961, p. 56]:

> A *thunk* is a piece of coding which provides an address. When executed, it leaves in some standard location (memory, accumulator, or index register, for example) the address of the variable with which it is associated. There is precisely one thunk associated with each actual parameter in each specific procedure statement . . . . If the actual parameter is a conditional expression, the thunk selects from the alternatives and delivers the appropriate address.
> In the most general case, the address transmitted by the thunk may be desired information, the address of the desired information, or the address where information is stored enabling the calculation of the desired information.

Since, in PAMS, "executable procedures" are expressed in a non-procedural language, the somewhat self-contradictory phrase "non-procedural thunk" becomes most apt.

It is through this principle that static templates become dynamic in nature. A single dynamic template thus defined ***a priori*** yields numerous, conceptually constant definitions when the template is applied to a given set of data. Interestingly, the principle also provides the mechanism for including "viewpoint dependencies" in PAMS. While the concept of user views or external schemas has become widespread in many commercially available relational database management systems, views are seldom, if ever, discussed in the literature as an integrated portion of a modeling system. They are not part of the Relational Model nor are they found in most semantic data models simply because the concept of a view does not seem to "fit together" with the other concepts found in such modeling systems. This is due to the fact that such dependencies form a special case of the more general feedback dependency

problem.

Since a properly formed semantic data modeling system should integrate application and database objects using a single representation, all view definitions are themselves objects in the application. If a view definition is a database object and the non-procedural thunk principle can be used to express the definitional dependency of one database object on the value of another, the expression of viewpoint dependencies simply becomes a special case of the former. This solution is predicated on the assumption that there is a suitable vehicle for expressing the definition of a view. Unlike a conventional system, one would not express each user view with a distinct, well-defined whole. Rather, the template of the database would undergo modification in as many places as necessary so that the viewpoint dependencies could be dynamically applied as the data was accessed.

4.4. Operational Dependency

Another major category of semantic dependencies arises from operations on the database. The list of such operations which might be applied to the Research Grant Example ranges from the ordinary to the esoteric. Some examples include:

Review a grant for funding.
Monitor a grant's progress.
Summarize allocated funds for an agency.
Call the telephone number of a principal investigator.
Find grants submitted multiple times.
Verify funding status of a grant.
Send a mailing to all principal investigators.
Determine the ethnic background of the principal investigators.
Verify that all institutions are eligible for funding.

The situation at hand is essentially the same as that addressed by behavioral models (as discussed in Chapter 2): the incorporation of knowledge about the operations being performed on the data into the application. The subtle difference here is a respect for the fact that this information may influence the "non-behavioral" aspects of the database description as well. In other words, the application is an *integrated* collection of both *object* and *operation* description information. In the terminology of the Prototype Activity Modeling System, this requires the application of *scripts* (operation descriptions) to *views* (object descriptions) so as to accomplish particular *tasks*.

Two basic problems with respect to this integration must be solved in order proceed further. The first is the method of expression for scripts and the second is the handling of operational dependencies which result from scripts. It is beyond the scope of this work to consider the best possible method for expressing operations in general. Scripts do not model operations in complete detail; since PAMS is not a simulation system, a complete, step-by-step recounting of the operation is not germane. For PAMS, like most semantic data modeling systems, only partial knowledge of the operation is needed to account for the effect of the operation on the database.

At this level of PAMS, an entity is either an object or an operation acting upon an object, but both share a common representation as an application object. Thus, the script, or operation, is an entity with a *role* that is special (but a description that is not). In past modeling systems operations and objects were viewed as distinct concepts and used abstract

algebras, pre- and post-conditions, petri nets, or similar methods for the representation of operations. Because it is a basic premise of PAMS to integrate all concepts within dynamic templates, most of these past methods are inappropriate as they do not integrate sufficiently with other application objects. Pre- and post-conditions are the exception, however, since they may be represented using the same SQL-style query specification language already introduced to express non-procedural thunks. In fact, pre- and post-conditions in PAMS are non-procedural thunks. While not necessarily a complete solution, this choice does allow a surprisingly large percentage of scripts to be represented. (Suitable enhancements to the query specification language should allow many more.)

For example, consider the net effect of a review script which might be applied to objects such as grants. The operation produces a result indicating that the object was either approved or denied:

REVIEW:
SELECT *
FROM {(approved), (denied)}
VIA EXTERNAL CONDITION

The fact that the selection criteria relies on an external condition indicates that this is, indeed, a script since the actual operation is external to the application and the script embodies only partial knowledge. At this point that knowledge consists only of a post-condition for the review script. For some scripts, even this amount of partial knowledge is more than required. The simplest possible script definition consists only of the the specification that the object is an external condition.

If pre-conditions also exist, such as the requirement that *REVIEW* be applied only to grant objects, the specification above would be expanded to:

REVIEW:
SELECT *
FROM {(approved), (denied)}
VIA EXTERNAL CONDITION
PROVIDED OBJECT IN *GRANT*

Perhaps the key aspect of script specification has been glossed over, however. Loosely speaking, the solution does not provide for any *alternatives* and seldom will a single non-procedural thunk suffice to represent a script. In the case of grants, the specification simplisticly suggests that there is a single script governing all reviews. One must realize, however, that the process of reviewing grants may vary from grant to grant and/or from agency to agency; in some sense, reviewing a computer science grant is different from reviewing a social science grant. Thus, as noted by Bradley [Bradley1978], the structure of an operation may depend on the data to which it is applied.

Under any realistic circumstance, *GRANT* must be reviewed conditionally: one does not review unreviewed grants, one reviews grants of a particular subtype that happen to be unreviewed grants. This distinction may seem superficial, but it has a significant impact on the way in which the operation is modeled. In the simpler case, a single *REVIEW* script is bound to each of the *GRANT*s unconditionally. When alternatives are considered, one of several possible review scripts is bound to a grant, but only on the condition that the type of grant and the type of review script are compatible. This is tantamount to adding an additional pre-condition for each type of review script. While this suggests that *REVIEW* scripts must be constructed for each category (and conceivably for each grant), it is preferable *apparently*

to change the script as the grant changes. This suggests that a *REVIEW* script is hierarchical in nature, containing nested non-procedural thunks as the means of implementing conditional definitions. When applied to the example of grant reviews, the nesting might be as follows:

REVIEW:
SELECT *
FROM {(computer science review), (social science review)}
VIA POSITION OF *DEPARTMENT*
IN {(computer science), (social science)}

COMPUTER SCIENCE REVIEW:
SELECT *
FROM {(approved), (denied)}
VIA EXTERNAL CONDITION
PROVIDED OBJECT IN *COMPUTER SCIENCE GRANT*

SOCIAL SCIENCE REVIEW:
SELECT *
FROM {(approved), (denied)}
VIA EXTERNAL CONDITION
PROVIDED OBJECT IN *SOCIAL SCIENCE GRANT*

This notion is sufficient, however, only if there is a name for each of the processes involved and if the list of such processes is constant over all applications of the database. In the above example, only computer science and social science reviews are included. Such is not the realistic case. The types of grants to be reviewed clearly involve feedback dependencies (such as which granting agencies are modeled), thus the types of grant review are also subject to these same feedback dependencies.

Once thunks become nested, however, the distinction between scripts and (ordinary) database objects begins to blur. Up to this point, a script has been principally identified with the notion of a dependency on external conditions. This is because scripts embody knowledge about operations being applied to the database (presumably by outside forces). When a script definition such as

REVIEW:
SELECT *
FROM {(computer science review), (social science review)}
VIA POSITION OF *DEPARTMENT*
IN {(computer science), (social science)}

is viewed in isolation, however, one cannot readily distinguish it from any other definition involving a feedback dependency such as

TELEPHONE:
SELECT *
FROM
{({area code, exchange, number}),
({country code, routing code, number}),
({country code, number})}
VIA POSITION OF *ADDRESS.COUNTRY*
IN {(US), (UK), (Panama)}

The reason is simply that there is no distinction between these types of objects. Scripts are roles for objects, not object types. Any object may actually be a script in this sense since the distinction is solely the result of how the object is perceived.

With the basic notions of script specification laid to rest, it is now possible to consider the impact of scripts on the definition of other application objects. It is clear that many operations applied to objects have an effect on the *apparent* definitions of those objects. Consider the definition of a *GRANT*. Regardless of what other attributes it may have, once it is considered that a grant may be reviewed, and that the result of such review is either approval or denial, there must be some way of storing that result. In a conventional database setting, this would mean the addition of an attribute to *GRANT* (possibly called *FUNDING DECISION*) in which to record the result. If other operations (such as monitoring of grant progress) are also considered, still more attributes would be added. Logically, none of these operations change what constitutes a grant as far as the real world is concerned, yet each requires a change in the definition of *GRANT* with respect to the overall application. Thus the ***structure*** of an object apparently depends on the way in which it is used, even though the corresponding real world object is unchanged.

In the Prototype Activity Modeling System a more consistent approach is taken. Since scripts are objects, it is possible to consider these operation dependent attributes as part of the definition of the ***script*** rather than of the object to which the script is applied. This approach preserves the integrity of object definitions when new operations are added. Of course there is a slight complication here in that the actual data value for such an attribute is not tied to the script ***per se*** but rather to the pairing of the script and its object. This is handled by considering each such value as a new instance of the script object in exactly the same way that a new *GRANT* instance is created when a new grant is entered into the database. Thus, in the case of grant reviews, the *FUNDING DECISION* is not tied to the *GRANT* but rather to the *REVIEW* created by the binding of that script to that object.

A method for specifying tasks should also now be apparent. Forming a task by binding a script to an object is simply a Cartesian aggregation of two application objects with particular roles: the object and the script.

GRANT REVIEW:
(*GRANT*, *REVIEW*)

The problem is actually more complicated, however. While a pairing of object and operation may uniquely identify a task at any point in time, in all likelihood the pairing will not uniquely identify the task over time. Once one has a view to which a script can be applied, that script will very possibly be applied to the view more than once. The methods examined thus far are sufficient only if the binding of a *REVIEW* script to a particular *GRANT* is a one time event. If, for instance, the grant is submitted ***several*** times (to the same agency) then several review/grant aggregations occur. Logically however, these separate bindings of *REVIEW* and *GRANT* all represent parts of the same single task — that of reviewing the grant. In this case, the entire review process consists of several review instances, but nevertheless, the task is a single entity.

Once again, the single value trap has surfaced. While one normally considers a grant review as a single application of *REVIEW* to *GRANT*, this is not always the case; there are exceptions. The solution is to consider all tasks as potentially multivalued collections of script/application object bindings. Such problems are not limited to multiple occurrences of the same binding. It is also possible for several ***distinct*** objects to be bound to the same script, yet ***not*** represent distinct tasks (as when several revisions of a grant are submitted). Analogously, several ***distinct*** scripts may be bound to a single object, yet ***not*** represent distinct tasks (as when the same grant is submitted to and reviewed by several different

agencies). In each case, logically, a single grant is undergoing a review.

The PAMS approach to these problems is based on the idea that bindings are dynamic, rather than static, constructs. The binding that enables any task may actually change over the life of that task as participants in the binding change due to other constraints. In other words, the script bound to some application object to perform a task is a ***bundle*** of aggregations rather than exactly one. It is this bundle that is the agent of change, not any one particular binding. Both scripts and application objects participating in the aggregation may come and go, yet the task itself is unaffected. Thus the previous example is more properly expressed:

GRANT REVIEW:
{(*GRANT*, *REVIEW*)}

If it is ***absolutely*** necessary to derive a single binding per task (because of the way the task information is being used), the ***value nomination principle*** may be used to derive a single value. However, it should be noted that this derivation may well depend on the intended usage: the most obvious nomination is the most recent review; within a given agency, however, the more logical choice may be the most recent review done by that agency.

4.5. State Dependency

Another major category of semantic dependencies arises because the objects which comprise the database are not a static collection. Since the collection changes (by adding/deleting/modifying objects), it is reasonable to assume that this ***change of state*** will provoke ***apparent*** change in the contents of other parts of the database. One aspect of state change which occurs in the Research Grant Example is due to the reviewing of grants for funding. All grants are initially "unreviewed." At some point in time, each particular grant transitions to an "under review" state, and still later to a "reviewed" state. In a conventional database setting, the result of such a state change would be recorded in an appropriate attribute, but only if the attribute and the recording mechanism were explicitly provided as part of the application design. Further, the reason for the change would be lost unless a behavioral model were being used. Even then, only a ***single*** status value would be contained in the database since, as noted much earlier, applications typically represent snapshots of data at (imaginary) fixed points in time. The ***change of state*** from one point in time to another is neither captured nor modeled.

Using the PAMS concepts presented in the previous section, such questions about state change are posed whether or not a script has been applied to form a task: for instance, whether a particular *GRANT* and a particular *REVIEW* have been aggregated together to produce *GRANT REVIEW*. This leads to the ***activity monitoring principle*** of PAMS. A dynamic record of all tasks is kept, may be accessed, and may cause apparent change in the definition of other application objects.

In the case of PAMS, tasks are the cause of change to the database and thus form the basis for activity monitoring. The solution to the review status problem might be expressed thusly:

REVIEW STATUS:
SELECT *
FROM {(unreviewed), (under review), (reviewed)}
VIA STATE OF *GRANT REVIEW*

It is also typical to base a dependency simply on the existence (or non-existence) of a particular task binding.

Any change must be measured with respect to some criteria: for instance, change is usually considered to occur *over time*. While task bindings are often dynamic with respect to the passage of time, time is arguably an abstraction which can be represented by an ongoing operation. (See, for example [Anderson1982] or [Bolour1983]). Thus, the Prototype Activity Modeling System approach considers each task as driven by a *clock* which is simply another task; this dynamic binding then constitutes a database *activity*.

The notion of a clock is used very broadly in PAMS. Not only can a task depend on points or intervals in some periodic abstraction of *time*, aperiodic clocking is also possible since any task may depend on clock signals sent by *any* other task. Any task may be used as a clocking task since only the role of the task (not its form or description) is special. A clock signal (in its simplest form) is nothing more than a post condition of the script bound to form the clocking task.

This view of task clocking is overly simplistic, however, since it is contrary to the normal connotation: a clock does not stop and produce a result. Intuitively, a clock is something which runs (seemingly forever) which may be viewed to determine the current "time" or state of affairs. Thus clock signals are not strictly post-conditions, but rather intermediate conditions which arise during the execution of the clock task. Fortunately, this poses no additional problem since each intermediate condition is logically the post-condition of a nested task.

Interestingly, it is through the activity monitoring principle that PAMS incorporates another important semantic database notion. Generalization hierarchies, as originally introduced by Smith and Smith [Smith1977b], form a major building block in many semantic data modeling systems, including Codd's RM/T. The Prototype Activity Modeling System, however, explicitly omits support for this form of generalization *per se*. As has already been noted in the preceding chapter, generalization is an error-prone concept because of the premature generalization traps which must be avoided. Further, while both Codd and the Smiths claim that generalization is an orthogonal principle to aggregation, it is this author's contention that the two may be unified in the same way that RM/T unifies application and database objects or in the way that the original Relational Model unifies entities and relationships.

The approach taken with PAMS can best be expressed by comparison. Rather than considering an object as a generalization of several sub-types, each sub-type object is considered a specialization of the former. By definition, at least as much, if not more, information is recorded in the application for a specialized object as would be for its more general counterpart. To specialize an entity, however, PAMS requires knowledge about the operation which specialized that object. (Thus, for a *GRANT* to be specialized into a *REVIEWED GRANT*, the grant must have been reviewed.)

In this way, it is perhaps more accurate to consider generalization as an operation which "dis-aggregates" the sub-type instance (*REVIEWED GRANT*) into its constituent objects: the specialization agent (*REVIEW*), the object of the specialization (*GRANT*), and any attributes relevant solely to the relationship itself. In other words, specialization establishes a relationship object and through this relationship a composite aggregation may be accessed; generalization inverts this operation. This form of generalization is far more restrictive than

that of Smith and Smith which might be termed *permissive generalization* since (1) it applies to all instances of the object type which have been generalized and (2) it encourages false generalizations. Inverse specialization, however, is perhaps best termed *pessimistic generalization* since no instance of an object may be generalized unless a specific instance of the binding operation is known.

On the surface it does not appear that all generalizations can be handled in this way, however, since they do not all result from operations on the database. Consider the often used example that a *BICYCLE* generalizes to a *LAND VEHICLE* which, in turn, generalizes into a *VEHICLE*. Such generalizations are designed into the application rather than derived from the "run-time" execution of operations. With PAMS, however, there is very little (if any) distinction between database design and run-time operation since all definitions (i.e., results of design time operations) are indistinguishable from other ordinary database objects. Thus, even these obvious generalizations can be thought of as the result of task execution.

Further, inverse specialization is more fundamental in nature than Smith and Smith's form of generalization since it is applied at the level of individual objects rather than object types. Oddly, Cartesian aggregation, which is also attributable to Smith and Smith, *is* fundamental in this same way. When aggregation is applied in the same way as Smith and Smith's generalization, the result is that *any* combination of objects (from the aggregated types) forms an aggregate object. For instance, using Smith and Smith's original example (see Section 3.2.1), any combination of a person, hotel, room, and date becomes a valid reservation. It is, of course, the selective nature of aggregation applied to individual objects which makes the concept useful. Since PAMS is based on aggregation, it is consistent that PAMS use the more fundamental form of generalization as well.

4.6. Interpreting the Approach

It is clearly possible to discuss many additional dependencies one might find in applications of an activity modeling system. However, this is an appropriate point to stop since it is now possible to summarize one approach for modeling database activity — the one embodied in the Prototype Activity Modeling System. If this approach is at all reasonable, then this section *must be short*. Recall from Chapters 1 and 2 that current semantic modeling systems (such as RM/T and SDM) are very complex and suffer from the lack of a reasonably simple interpretation. Such interpretations are effectively concise conceptual summaries.

For the Prototype Activity Modeling System, the interpretation can best be expressed as the near homogeneous blending of all operations and objects into a forest of varying, self-defining trees or graphs. This in turn can be expressed in more detail as a four step design and implementation process. The steps are not completely distinct, however, since the concept of database activity deliberately blurs many of the differences.

(1) Determine the set of objects involved in the application and group these together in as many containment (aggregation) hierarchies as necessary. This step accounts for most traditional notions of database structure as well as some features usually found only in semantic models (such as cover aggregations).

(2) Determine the set of operations involved in the application and group these together in as many additional containment (aggregation) hierarchies as necessary. This step accounts for behavioral information in a way which deliberately blurs the distinction between operations and objects.

(3) Intertwine the hierarchies as necessary to represent the application of operations to objects. This step accounts for the definition of tasks as well as for generalizations, state dependencies, and inter-task synchronization dependencies through the use of the *activity monitoring principle.*

(4) Modify the hierarchies to include alternative definitions. This step accounts for a diverse group of dependencies in a unified way. User views, simple behavioral relationships, and feedback dependencies are all handled by conditional template construction using the *non-procedural thunk principle.* Seldom is this last step actually applied totally after the first three; usually a designer will add alternative definitions as their need arises.

There are detailed issues remaining to be answered within each of these activity modeling steps. The framework and basic principles contained within the Prototype Activity Modeling System establish *one* possible solution (not the only one nor necessarily the best one). In the next chapter the application of this prototype framework is pursued through an in-depth example and in Chapter 6 the pros, cons, and future directions are weighed.

CHAPTER 5

A Comparative Example

This chapter applies the Prototype Activity Modeling System to a real-world design problem: the *IFIP Working Conference Problem.* Since the problem was created as a basis for comparing different approaches to information systems design, numerous solutions have been published in recent years. This affords the opportunity to apply PAMS in a way that facilitates comparison to other approaches.

5.1. The IFIP Working Conference Problem

Without a common basis, it is difficult to compare different data modeling approaches. Since authors tend to pick examples which highlight the benefits and hide the deficiencies of their own models, almost any comparison based strictly on the published literature would be inadequate. Similar problems have been noted in other areas of computer science and in particular, by the IFIP Working Group on the Design and Evaluation of Information Systems (WG 8.1). Beginning in 1979 at the suggestion of T. William Olle, an effort was made by the Working Group to define a "standard test case" over which many different information system design methodologies reasonably could be compared. Because of the closely related nature between information systems design and data modeling, the IFIP Working Conference Problem (as the test case has come to be called) may also be used to compare data modeling techniques. The complete problem definition, as taken from [Olle1982a, appendix 1], is reproduced below.

IFIP

COMPARATIVE REVIEW OF
INFORMATION SYSTEMS DESIGN METHODOLOGIES

Problem Definition

1. Background

An IFIP Working Conference is an international conference intended to bring together experts from all IFIP countries to discuss some technical topic of specific interest to one or more IFIP Working Groups. The usual procedure, and that to be considered for the present purposes, is an invited conference which is not open to everyone. For such conferences it is something of a problem to ensure that members of the involved IFIP Working Group(s) and Technical Committee(s) are invited even if they do not come. Furthermore, it is important to ensure that sufficient people attend the conference so that the financial break-even point is reached without exceeding the maximum dictated by the facilities available.

IFIP Policy on Working Conferences suggest [*sic*] the appointment of a Program Committee to deal with the technical content of the conference and an Organising Committee to handle financial matters, local arrangements, and invitations and/or publicity. These committees clearly need to work together closely and have a need for common information

and to keep their recorded information consistent and up to date.

2. Information system to be designed

The information system which is to be designed is that necessary to support the activities of both a Program Committee and an Organising Committee involved in arranging an IFIP Working Conference. The involvement of the two committees is seen as analogous to two organizational entities within a corporate structure using some common information.

The following activities of the committees should be supported.

Program Committee:

1. Preparing a list to whom the call for papers is to be sent.
2. Registering the letters of intent received in response to the call.
3. Registering the contributed papers on receipt.
4. Distributing the papers among those undertaking the refereeing.
5. Collecting the referees' reports and selecting the papers for inclusion in the program.
6. Grouping selected papers into sessions for presentation and selecting chairman for each session.

Organising Committee:

1. Preparing a list of people to invite to the conference.
2. Issuing priority invitations to National Representatives, Working Group members and members of associated working groups.
3. Ensuring all authors of each selected paper receive an invitation.
4. Ensuring authors of rejected papers receive an invitation.
5. Avoiding sending duplicate invitations to any individual.
6. Registering acceptance of invitations.
7. Generating finallist [*sic*] of attendees.

3. Boundaries of system

It should be noted that budgeting and financial aspects of the Organising Committee's work, meeting plans of both committees, hotel accommodation for attendees and the matter of preparing camera ready copy of the proceedings have been omitted from this exercise, although a submission may include some or all of these extra aspects if the authors feel so motivated.

It must be emphasized that even though at least thirteen "solutions" to the Working Conference Problem have been published [Olle1982b], the results derived in this work *cannot* be compared directly with other solutions. The Prototype Activity Modeling System, or for that matter, any activity modeling system, is not an information system design methodology. Like any other database system applied in this context, PAMS only *supports* the design methodology by capturing and "implementing" parts of the design.

5.2. Requirements Analysis

The IFIP Working Conference Problem appears deceptively simple on first reading. The problem, however, leaves room for many interpretations to cover different cases and perceptions [Verrijn-Stuart1982, p. vi]:

> Even a superficial consideration will reveal that, whilst it certainly is not an over-complex case, it is very far from trivial. The reaction of the TC8 [parent Technical Committee of WG 8.1] family varied enormously. Some thought it so well defined that the problem statement, in their view, amounted to a full systems requirement definition, i.e. there seemed nothing more to be "designed." On the other hand there were those who considered it so ill-structured that an extensive formalization would be required before the first design step might be dreamt of. . . .

As far as this work is concerned, such varied interpretations are both a boon for exposition and a cause for immediate concern. The less than rigid specification of the problem works to advantage because PAMS is designed to be flexible and cope with alternative definitions. Thus, many of PAMS basic benefits can be illustrated by incorporating problem variations within a solution. Still, PAMS requires a fairly detailed problem specification before it can be applied. It provides no explicit support for *requirements analysis*, an early step in the design of any information system, which resolves many variations and fills-in missing details. For the task at hand, such analysis will be done on a relatively *ad hoc* basis, borrowing from various methodologies, such as Sysdoc [Aschim1982], D2S2 [MacDonald1982], and NIAM [Verheijen1982], where desirable.

As a first step in the analysis of the Working Conference Problem, nouns and noun phrases have been extracted from the problem description and then categorized in zero or more of the following ways to build the informal equivalent of an aggregation and generalization hierarchy:

(1) *attribute* of an object,
(2) *specialization* of an object,
(3) *synonym* for another noun.

The nouns and their categorizations are shown in Tables 3 through 6 which correspond to the three sections of the problem description plus those entries not in the description but which are implied by it.

The association of these categories with each noun represents a series of design decisions which are not necessarily optimal nor even correct. (As discussed later, some are obviously incorrect.) It should be noted that the nouns have been deliberately and somewhat subjectively edited to eliminate duplicates and irrelevant phrases of a general nature (such as "common information"). Note also that each list is ordered by occurrence of the noun or noun phrase within the problem and that several abbreviations are used due to space limitations:

IFIP Working Conference	specialization of conference
expert	synonym for person
IFIP country	attribute of IFIP
technical topic	attribute of IFIP Working Conference
IFIP WG	attribute of IFIP
invited conference	specialization of IFIP Working Conference
conference	(none)
member of involved IFIP WG	attribute of involved IFIP WG specialization of person
involved IFIP WG	specialization of IFIP WG attribute of IFIP Working Conference
member of involved IFIP TC	attribute of involved IFIP TC specialization of person
involved IFIP TC	specialization of IFIP TC attribute of IFIP Working Conference
person	(none)
financial break-even point	attribute of financial matter
maximum (capacity)	attribute of facility
facility	attribute of local arrangement
IFIP Policy	attribute of IFIP
PC	specialization of committee
technical content of conference	attribute of PC's work
OC	specialization of committee
financial matter	attribute of OC's work
local arrangement	attribute of OC's work
invitation	attribute of OC's work
publicity	attribute of OC's work

Table 3. Noun Phrases from Section 1 of the Working Conference Problem

TC for Technical Committee, WG for Working Group, OC for Organising Committee,[1] and PC for Program Committee.

As far as the behavioral aspects of the Working Conference Problem are concerned, the way in which the problem description is presented practically generates its own "first cut" of necessary tasks. Each of the committee activities is an obvious candidate though some logically break into several distinct tasks as, for example, the last activity of the Program Committee:

(1) groups papers into sessions, and
(2) selects a chairman for each session.

In addition, at least one of the activities is perhaps best thought of as a constraint rather than a task: the Organising Committee must ensure that no duplicate invitations are sent. With some rephrasing, an initial list of tasks emerges as shown in Table 7. In accordance

[1] The British spelling of "organising" is used throughout this chapter in order to remain consistent with the definition of the Working Conference Problem.

involvement of committee	synonym for committee's work
activity of committee	attribute of committee's work
list (to be sent call for papers)	specialization of list
call for papers	attribute of technical content of conference
letter of intent	(none)
contributed paper	(none)
referee report	(none)
program	attribute of technical content of conference
selected paper	specialization of contributed paper
	attribute of session
session	attribute of program
chairman	attribute of session
list of people (to invite)	(none)
priority invitation	specialization of invitation
National Representative	attribute of IFIP Country
	specialization of person
(Sponsor) WG member	specialization of member of involved IFIP WG
member of associated WG	specialization of member of involved IFIP WG
author of selected paper	specialization of author
	attribute of selected paper
author of rejected paper	specialization of author
	attribute of rejected paper
duplicate invitation	specialization of invitation
individual	synonym for person
acceptance of invitation	(none)
list of attendees	(none)

Table 4. Noun Phrases from Section 2 of the Working Conference Problem

budgeting aspect	attribute of financial matter
financial aspect	attribute of financial matter
OC's work	attribute of OC
	specialization of committee work
meeting plans of committee	attribute of committee
hotel accommodation	attribute of local arrangement
camera ready copy	attribute of proceedings

Table 5. Noun Phrases from Section 3 of the Working Conference Problem

IFIP	(none)
IFIP TC	attribute of IFIP
name (of country)	attribute of IFIP country
committee	attribute of IFIP Working Conference
PC's Work	attribute of PC
	specialization of committee work
committee's work	attribute of committee
author	specialization of person
	attribute of contributed paper
name	attribute of person
address	attribute of person
attendee	attribute of list of attendees
	specialization of person
people to invite	attribute of list (of people to invite)
	specialization of person
call for paper receiver	specialization of person
possible priority invitee	specialization of invitee
possible ordinary invitee	specialization of invitee
rejected paper	specialization of paper
(sponsor) WG	specialization of involved WG
associated WG	specialization of involved WG
proceedings	attribute of program
general chairman	attribute of IFIP Working Conference
	specialization of person
chairman of committee	attribute of committee
	specialization of person
title	attribute of contributed paper
member of committee	attribute of committee

Table 6. Noun Phrases Implied in the Working Conference Problem

with the PAMS approach, the list can be supplemented with additional tasks, Table 8, which specialize objects represented in the noun lists.

Thus, with this somewhat more specific problem statement, the next few sections follow the PAMS activity modeling approach discussed at the end of the previous chapter and summarized here:

(1) Specify containment hierarchies for objects.
(2) Specify containment hierarchies for operations.
(3) Intertwine the hierarchies to account for interactions.
(4) Add alternative definitions to the hierarchies.

5.3. The Object Hierarchies

Since the description of the Working Conference Problem centers on the "activities" of the Program and Organising Committees, one might characterize the problem statement as operation rather than object oriented. Thus the requirement analysis noun list is central to

Issue call for papers.
Register letter of intent.
Register contributed paper.
Issue paper to referee.
Register referee report.
Review paper.
Group papers for session.
Determine session chairman.
Prepare invitee list.
Issue priority invitation.
Issue invitation to selected author.
Issue invitation to rejected author.
Register acceptance.
Prepare attendee list.

Table 7. Tasks Which Result From Committee Activities

Call IFIP Conference.
Call international conference.
Determine style of IFIP Working Conference.
Determine associated Working Group.
Establish Program Committee.
Establish Organising Committee.
Review referee reports of paper.
Determine priority invitees.
Determine sponsor Working Group.

Table 8. Tasks Which Result From Object Specialization

the identification of objects and their hierarchies. Virtually all entries in the list represent objects of one kind or another. For this step of the PAMS approach, however, only "simple" objects and their hierarchies are relevant. Primary consideration is given to those entries categorized as attributes of other objects since they represent the Cartesian aggregation containment hierarchies. (Note that definitions for "leaf" elements of the containment hierarchies and definitions of isolated objects are not shown.)

IFIP:
{(Technical Committee, Working Group, Policy, Country)}

COUNTRY:
{(name, National Representative)}

IFIP WORKING CONFERENCE:
{(title, technical topic, general chairman,
involved Technical Committee, involved Working Group, committee)}

INVOLVED TECHNICAL COMMITTEE:
{(member)}

INVOLVED WORKING GROUP:
{(member)}

COMMITTEE:
{(chairman, member, work, meeting plans)}

WORK, INVOLVEMENT:
{(activity)}

ORGANISING COMMITTEE:
{(work)}

ORGANISING COMMITTEE.WORK:
{(financial matter, local arrangement, invitation, publicity)}

FINANCIAL MATTER:
{(budget aspect, financial aspect, financial break-even point)}

LOCAL ARRANGEMENT:
{(facility, hotel accommodation)}

FACILITY:
{(maximum capacity)}

PROGRAM COMMITTEE:
{(work)}

PROGRAM COMMITTEE.WORK:
{(technical content of conference)}

TECHNICAL CONTENT OF CONFERENCE:
{(call for papers, program)}

PROGRAM:
{(proceedings, session)}

PROCEEDINGS:
{(camera ready copy)}

SESSION:
{(chairman, selected paper)}

CONTRIBUTED PAPER:
{(author, title)}

SELECTED PAPER:
{(author)}

REJECTED PAPER:
{(author)}

PERSON, EXPERT, INDIVIDUAL:
{(name, address)}

ASSOCIATED WORKING GROUP:
{(member)}

SPONSOR WORKING GROUP:
{(member)}

LIST OF PEOPLE TO INVITE:
{(person to invite)}

LIST OF ATTENDEES:
{(attendee)}

As with most applications of PAMS (but unlike the deliberately chosen example of Chapter 4), the "forest" of hierarchy "trees" in the Working Conference Problem is relatively broad and disconnected at this point. The primary reason for this is that specialization objects are not accounted for until the third step of the PAMS approach (since they result from object/operation interactions). Once, for instance, the specialization of *COMMITTEE* into *ORGANISING COMMITTEE* and *PROGRAM COMMITTEE* is defined, the trees become much smaller in number and taller in height.

5.4. The Operation Hierarchies

Each of the tasks considered in the requirements analysis represents a binding of script and object. These scripts represent the operations involved in the Working Conference Problem. Defining the list of such scripts is somewhat of a "chicken and egg" problem, however, since the scripts and tasks are so closely related. Following the PAMS interpretation steps, however, one first defines scripts and then binds these to form tasks.

The simple solution is to *assume* that each task type involves a different script. Doing so fails to extract potential regularities in the application, however. For example, "issue invitation to selected author" and "issue invitation to rejected author" should share the use of a common *INVITE* script. Even more diverse tasks may share scripts at higher levels of abstraction; both "issue call for papers" and "issue invitation to selected author" could share an *ISSUE* script. Such design decisions substantially reduce the number of scripts involved in an application but make the development of the containment hierarchy somewhat more complex. One such possible distillation of scripts in the Working Conference Problem is simply:

ISSUE:
SELECT *
FROM *PERSON*
VIA EXTERNAL CONDITION

REGISTER:
SELECT *
FROM *PERSON*
VIA EXTERNAL CONDITION

IDENTIFY:
SELECT *
FROM {(name)}
VIA EXTERNAL CONDITION

REVIEW:
SELECT *
FROM *DECISION*:
{(select), (reject)}
VIA EXTERNAL CONDITION

DETERMINE PERSON:
SELECT *
FROM *PERSON*
VIA EXTERNAL CONDITION

GROUP PAPERS:
SELECT *
FROM *SELECTED PAPER*
VIA EXTERNAL CONDITION

SELECT REFEREE:
EXTERNAL CONDITION

INVOLVE:
SELECT *
FROM {(sponsor), (associate)}
VIA EXTERNAL CONDITION

SPONSOR:
EXTERNAL CONDITION

ASSOCIATE:
EXTERNAL CONDITION

PREPARE:
EXTERNAL CONDITION

DETERMINE STYLE:
SELECT *
FROM {(invited)}
VIA EXTERNAL CONDITION

Very few of the operations involved in the Working Conference Problem seem to produce any result; thus few of the script definitions have explicit post-conditions and there is little to distinguish one script from another. Further, very little hierarchical structure is exhibited by the scripts. Both these characteristics arise because the script encompassing tasks and the specialization objects have yet to be defined. As will be seen, such definitions knit much of the Working Conference Problem together.

It should be noted that the definition of the *REVIEW* script incorporates the definition of an *object* within the script. It is not necessary to give such an attribute a name, but in this case *DECISION* will be referenced later in other definitions. Thus the name serves a purpose. Likewise, it is not necessary to give the definition of *DECISION* within the definition of *REVIEW*; it could be given separately.

5.5. Intertwined Definitions

Most of the interconnections between object and operation hierarchies are relatively simple. They involve only the binding of scripts to objects so as to form tasks revealed during requirements analysis. Some of these are tasks which correspond directly to committee activities:

ISSUE CALL FOR PAPERS:
{(issue, call for papers)}

REGISTER LETTER OF INTENT:
{(register, letter of intent)}

REGISTER CONTRIBUTED PAPER:
{(register, contributed paper)}

GROUP PAPERS FOR SESSION:
{(group papers, session)}

SELECT CHAIRMAN:
{(determine person, session)}

PREPARE INVITEE LIST:
{(prepare, list of people to invite)}

PREPARE ATTENDEE LIST:
{(prepare, list of attendees)}

A few of the committee activity tasks are more complex, however, because they involve state dependencies on other tasks and/or feedback dependencies between the script and the object. An example which contains both types of dependencies is the definition of *ISSUE PAPER TO REFEREE*. One pre-condition must be added to make sure the "issuee" is eligible to be a referee (a state dependency on the task which defines *REFEREE*) and another to prevent the author from being issued his/her own paper to referee (a feedback dependency between *ISSUE* and *CONTRIBUTED PAPER*). Further, at least one of the definitions, *REVIEW PAPER*, highlights the somewhat inadequate nature of SQL as a vehicle for expressing pre-conditions. Logically, the necessary condition is quite simple: all referee reports must be in before the paper is reviewed for selection/rejection. The syntax of SQL makes the expression of the condition unnecessarily complex, however.

ISSUE PAPER TO REFEREE:
SELECT *
FROM {(issue, contributed paper)}
PROVIDED *ISSUE.PERSON* NOT = *CONTRIBUTED PAPER.AUTHOR*
AND *ISSUE.PERSON* IN *REFEREE*

REVIEW PAPER:
SELECT *
FROM {(review, contributed paper)}
PROVIDED
(SELECT COUNT(*)
FROM *REGISTER REFEREE REPORT*
WHERE *CONTRIBUTED PAPER*
= *REVIEW PAPER.CONTRIBUTED PAPER*)
=
(SELECT COUNT(*)
FROM *ISSUE PAPER TO REFEREE*
WHERE *CONTRIBUTED PAPER*
= *REVIEW PAPER.CONTRIBUTED PAPER*)

REGISTER ACCEPTANCE:
SELECT *
FROM {(register, acceptance of invitation)}
PROVIDED *REGISTER.PERSON* IN *INVITEE*

REGISTER REFEREE REPORT:
SELECT *
FROM {(register, referee report, issue paper to referee)}
PROVIDED *REGISTER.PERSON* NOT = *ISSUE PAPER TO REFEREE.PERSON*

Perhaps the most complex object/operation interaction in the Working Conference Problem arises from the activities which issue invitations. Recall that all members of the following categories are to be invited:

national representatives,
working group members,
members of associated working groups,
authors of selected papers,
authors of rejected papers, and
other people (as selected to meet financial break-even point).

Also recall that there are two additional constraints: each member of the first three categories receives a *PRIORITY INVITATION* (rather than an *ORDINARY INVITATION*) and no one is to receive a duplicate invitation. By mentioning duplicate removal, the problem implies that all categories must be direct or indirect subtypes of the same supertype. If this were not the case, there would be no basis for defining what was a duplicate. Thus, the problem demands that ***people*** be invited on a conditional basis: one does not invite national representatives, one invites people who happen to be national representatives. This represents a "classic" application of a non-procedural thunk, with *ISSUE INVITATION* and the necessary specializations defined as:

ISSUE INVITATION:
{(issue,
SELECT *
FROM {(priority invitation), (ordinary invitation)}
VIA FIRST POSITION OF *ISSUE.PERSON*
IN {(possible priority invitee), (possible ordinary invitee)})}

POSSIBLE PRIORITY INVITEE:
SELECT *
FROM *PERSON*
PROVIDED *PERSON*
IN UNBUNDLE
{national representative,
sponsor working group member,
associated working group member}

POSSIBLE ORDINARY INVITEE:
SELECT *
FROM *PERSON*
PROVIDED *PERSON*
IN UNBUNDLE
{selected paper.author,
rejected paper.author,
person to invite}

This approach *avoids* generation of duplicates rather than more conventional approaches which *eliminate* them after the fact. Hence it is not necessary to define a specialization for *DUPLICATE INVITATION* since the result would always be an empty bundle. Note also that some of the tasks identified during requirements analysis are actually specializations of *ISSUE INVITATION*:

ISSUE PRIORITY INVITATION:
SELECT *
FROM *ISSUE INVITATION*
WHERE *INVITATION* IN *PRIORITY INVITATION*

ISSUE INVITATION TO SELECTED AUTHOR:
SELECT *
FROM *ISSUE INVITATION*
WHERE *ISSUE.PERSON* IN *SELECTED PAPER.AUTHOR*

ISSUE INVITATION TO REJECTED AUTHOR:
SELECT *
FROM *ISSUE INVITATION*
WHERE *ISSUE.PERSON* IN *REJECTED PAPER.AUTHOR*

Returning to more general issues, many of the specialization objects revealed during requirements analysis are also produced by simple bindings. Some of these, such as *ORGANISING COMMITTEE*, include non-procedural thunks to select limited instances of the scripts or objects involved. Since these thunks operate on bundles, it is possible for the same instance to be specialized in more than one way (as opposed to other schemes which partition the instances into disjoint sets). For instance, a small conference could use a single committee to act as both the Program Committee and the Organising Committee.

IFIP WORKING CONFERENCE:
{(sponsor, IFIP)}

INVITED CONFERENCE:
{(SELECT *
FROM *DETERMINE STYLE*
WHERE ATTRIBUTE = 'invited',
IFIP Working Conference)}

INVOLVED IFIP WORKING GROUP:
{(involve, IFIP Working Group)}

INVOLVED IFIP TECHNICAL COMMITTEE:
{(involve, IFIP Technical Committee)}

SPONSOR WORKING GROUP:
SELECT *
FROM {(sponsor, involved IFIP Working Group)}
PROVIDED *INVOLVED IFIP WORKING GROUP*
NOT IN *ASSOCIATED WORKING GROUP*

ASSOCIATED WORKING GROUP:
SELECT *
FROM {(associate, involved IFIP Working Group)}
PROVIDED *INVOLVED IFIP WORKING GROUP*
NOT IN *SPONSOR WORKING GROUP*

PROGRAM COMMITTEE:
{(SELECT *
FROM *IDENTIFY*
WHERE *NAME* = 'Program',
committee)}

ORGANISING COMMITTEE:
{(SELECT *
FROM *IDENTIFY*
WHERE *NAME* = 'Organising',
committee)}

PERSON TO INVITE:
{(identify, person)}

SELECTED PAPER:
{(contributed paper,
SELECT *
FROM *REVIEW*
WHERE *DECISION* = 'select')}

REJECTED PAPER:
{(contributed paper,
SELECT *
FROM *REVIEW*
WHERE *DECISION* = 'reject')}

PRIORITY INVITATION:
{(SELECT *
FROM *IDENTIFY*
WHERE *NAME* = 'Priority',
invitation)}

ORDINARY INVITATION:
{(SELECT *
FROM *IDENTIFY*
WHERE *NAME* = 'Ordinary',
invitation)}

Some specializations do not need to be explicitly defined since their existence is implied by other definitions. Take, for instance, the *AUTHOR* of a *SELECTED PAPER*. From the original requirements analysis, this was identified as both a specialization of author and an attribute of contributed paper. Since selected paper is a specialization of contributed paper, the following definitions (which have already been presented) imply that each selected paper has an author:

CONTRIBUTED PAPER:
{(author, title, referee report)}

SELECTED PAPER:
{(contributed paper,
SELECT *
FROM *REVIEW*
WHERE *DECISION* = 'select')}

Thus it is perfectly correct to reference *SELECTED PAPER.AUTHOR* without further definition. This type of situation accounts for each of the following specializations:

REJECTED PAPER.AUTHOR
SPONSOR WORKING GROUP.MEMBER
ASSOCIATED WORKING GROUP.MEMBER

In a very similar way, some specializations are actually synonyms for particular attributes of task definitions:

CALL FOR PAPER RECEIVER:
ISSUE CALL FOR PAPERS.PERSON

AUTHOR:
REGISTER CONTRIBUTED PAPER.PERSON

ATTENDEE:
REGISTER ACCEPTANCE.PERSON

GENERAL CHAIRMAN:
SELECT *PERSON*
FROM {(determine person, IFIP Working Conference)}

CHAIRMAN:
SELECT CHAIRMAN.PERSON

LIST TO BE SENT CALL FOR PAPERS:
BUNDLEUP *ISSUE CALL FOR PAPERS.PERSON*

The definition of *GENERAL CHAIRMAN* is particularly interesting since the task involved in the specialization is defined within the specialization above. This unnamed task, which could be called *SELECT GENERAL CHAIRMAN*, clearly is a required part of any Working Conference Problem solution, yet it was not identified during the requirements analysis phase of the design.

A few of the specialization definitions given above conflict with object or operation definitions given earlier in the original construction of the hierarchies. For example, *ORGANISING COMMITTEE* has now been defined two ways:

ORGANISING COMMITTEE:
{(work)}

ORGANISING COMMITTEE:
{(SELECT *
FROM *IDENTIFY*
WHERE *NAME* = 'Organising',
committee)}

Since the second definition, the specialization, is now understood, the original definition is no longer necessary nor appropriate. This is not to say that the *WORK* attribute no longer exists, but rather that *ORGANISING COMMITTEE.WORK* is either *IDENTIFY.WORK* or *COMMITTEE.WORK*. In more general terms (following the PAMS approach) attributes of a specialization are usually inherited from either the script or the object used to derive the specialization. As a design decision, it is most appropriate in this case to allocate *WORK* to *COMMITTEE* since the existence of *COMMITTEE.WORK* was already established during requirements analysis. It seems that all such redefinitions can, indeed, be avoided by more detailed requirements analysis, but such deliberately was not done in this chapter. The following object definitions (some of which are used as scripts) must replace definitions presented earlier:

WORKING GROUP:
{(member)}

TECHNICAL COMMITTEE:
{(member)}

SPONSOR:
SELECT *
FROM *CONFERENCE*
VIA EXTERNAL CONDITION

ASSOCIATE:
SELECT *
FROM *CONFERENCE*
VIA EXTERNAL CONDITION

When an object is both a specialization of a second object as well as an attribute of a third, seemingly circular definitions can result. Consider the case of *AUTHOR* which is a specialization of *PERSON* and an attribute of *CONTRIBUTED PAPER*. The relevant definitions are:

CONTRIBUTED PAPER:
{(title, author)}

AUTHOR:
{(register contributed paper, person)}

REGISTER CONTRIBUTED PAPER:
{(register, contributed paper)}

While these definitions appear circular, they are not. The definitions merely give names to convenient sub-components of an important (though nameless) aggregation which represents a more fundamental way of defining the "register contributed paper" activity of the Program Committee: (register, title, person). It is the interaction of this activity with other activities such as "issue invitation to selected author" which generates the need (during requirements analysis) for the additional categories defined above.

5.6. Adding Alternatives

Given that the Working Conference Problem is so subject to interpretation, a large number of design decisions can be made in more than one "correct way." Some of these decisions are nothing more than judgement calls on the part of the designer, yet others hinge on the availability of additional knowledge (beyond the problem statement) or on the breadth of situations to which the resulting design may be applied. The dynamic template structures of PAMS allow such decision processes to be captured in the application description, thus deferring the decisions from "design time" to "run-time." The result is a far more flexible design.

In this chapter, to present all possible alternative definitions of objects, operations, and their hierarchies is both unnecessary and unwise. No real application of PAMS would defer all possible alternatives to run-time consideration since many *possible* alternatives are actually irrelevant or incorrect. Further, the open nature of the problem statement (and lack of a user "approved" requirements definition) multiplies the number of alternatives. Thus only a representative sampling is pursued in this section.

As a first example, consider that with further knowledge it is more appropriate for *IFIP WORKING GROUP* to be considered an attribute of an *IFIP TECHNICAL COMMITTEE* rather than an attribute of *IFIP* directly. This alternative pair of aggregations more accurately reflects the actual structure of IFIP, yet replacement of the original definitions of *IFIP* and *IFIP TECHNICAL COMMITTEE* (with new definitions which conform only to the more enlightened view) yields an inaccurate reflection of the problem statement. The best solution (from a design standpoint) is to allow for both possibilities and choose between them based on an external condition because even though the enlightened view clearly is correct, seldom will such disagreements be so easily resolved in a real world design. Thus, the design may be held open pending further information by modifying the definitions of both objects to include alternative aggregations:

```
SELECT *
FROM
   {(IFIP:
      {(Technical Committee, Working Group, Country, Policy)},
    TECHNICAL COMMITTEE:
      {(number, title, purpose, member)} ),
   (IFIP:
      {(Technical Committee, Country, Policy)},
    TECHNICAL COMMITTEE:
      {(number, title, purpose, member, Working Group)})}
VIA EXTERNAL CONDITION
```

It could be appropriate to incorporate a distinction between an *INVITED CONFERENCE* and an *INVITED PAPER CONFERENCE* where the only persons eligible to contribute papers are those who received an *INVITATION TO SUBMIT* (which is simply a synonym for *CALL FOR PAPERS*). In this case a new conference style must be added to the *DETERMINE STYLE* script and a pre-condition must be added to the *REGISTER CONTRIBUTED PAPERS* task to allow contributions only from those who have received the *CALL FOR PAPERS* if the conference is of this type.

```
INVITATION TO SUBMIT:
   CALL FOR PAPERS
```

DETERMINE STYLE:
SELECT *
FROM {(invited), (invited paper)}
VIA EXTERNAL CONDITION

REGISTER CONTRIBUTED PAPER:
SELECT *
FROM {(register, contributed paper)}
PROVIDED
(SELECT *
FROM {(), (register.person in invitation to submit.person)})
VIA POSITION OF *CONFERENCE STYLE* IN {(invited), (invited paper)})

While it may look as though *DETERMINE STYLE* has been redefined here, such is not the case. The change would be accomplished by simply inserting the new value into the bundle from which the style is selected.

Similar logic can be used to provide numerous other options. One such option could limit the submission of a *CONTRIBUTED PAPER* to people who previously submitted a *LETTER OF INTENT*. Another option could require that a *MEMBER* of the *PROGRAM COMMITTEE* be a *MEMBER* of the *SPONSOR WORKING GROUP*.

It is also possible to define *ATTENDEE* in a large variety of ways. It could be required, for instance, that a person must actually attend the conference (rather than just accept the invitation) in order to be considered an *ATTENDEE*. As another alternative it could be required that the Organising Committee "approve" people who have accepted (in order to assure that facility capacity is not overrun). Further, there is no reason these two requirements could not be combined to form a third alternative to the original simple definition of *ATTENDEE*. (Note that the following definition of *ACCEPTOR* is identical to the original definition of *ATTENDEE*; in effect another level of specialization has been created.)

ATTEND:
EXTERNAL CONDITION

ATTENDEE:
{(SELECT *
FROM {(register acceptance, attend)}
VIA EXTERNAL CONDITION,
SELECT *
FROM {(invitee, approved acceptor)}
VIA EXTERNAL CONDITION)}

APPROVED ACCEPTOR:
{(approve, acceptor)}

APPROVE:
EXTERNAL CONDITION

ACCEPTOR:
REGISTER ACCEPTANCE.PERSON

Even though not directly stated in the problem description, there are other ways to define *CALL FOR PAPER RECEIVER*. "The call may be associated with a specific person or may be a general request to [an IFIP] member society or even a published advertisement."[MacDonald1982, p. 265] Since the existing definition

CALL FOR PAPER RECEIVER:
ISSUE CALL FOR PAPERS.PERSON

merely identifies it as an attribute of a task, incorporating such alternatives also requires that the definition of *ISSUE CALL FOR PAPERS* be revised:

CALL FOR PAPER RECEIVER:
ISSUE CALL FOR PAPERS.{(person), (pseudo person)}

ISSUE CALL FOR PAPERS:
{(SELECT *
FROM {(issue, broad issue)}
VIA EXTERNAL CONDITION,
call for papers)}

BROAD ISSUE:
SELECT *
FROM *PSEUDO PERSON*
VIA EXTERNAL CONDITION

PSEUDO PERSON:
UNBUNDLE {person, publication, IFIP.member society}

IFIP:
{(Technical Committee, Working Group, country, policy, member society)}

So far, the solution of the Working Conference Problem has assumed that each *PRIORITY INVITATION* is the same and that the priority invitation does not depend on why the person is being invited (so long as one is a member of the groups to which the priority invitation operation can be applied). Of course, such is not necessarily the case. The invitation a *NATIONAL REPRESENTATIVE* receives may be different from that received by a *SPONSOR WORKING GROUP MEMBER* and still different from that for an *ASSOCIATED WORKING GROUP MEMBER*, yet all may be *PRIORITY INVITATION*s. Even within this specialization, two further alternatives stand out: (1) custom invitations could mention all the reasons a particular person is being invited, or (2) a priority scheme could be established to mention only the highest priority reason. Either alternative requires the use of bundle ordering. To implement the first approach, order the underlying bundle of reasons for priority invitation and select the appropriate *PRIORITY INVITATION* based on the reasons which apply to any particular *PERSON*. For the second approach, further refine the definition of *PRIORITY INVITATION* to use only the *first* script in each bundle. Thus, each *PERSON* will receive the invitation of the highest applicable priority. Using two external conditions to subdivide the alternatives, the appropriate definitions might be as follows:

PRIORITY INVITATION:
SELECT *
FROM {(specialized priority invitation), (old priority invitation)}
VIA EXTERNAL CONDITION

SPECIALIZED PRIORITY INVITATION:
SELECT *
FROM
{(national representative),
(sponsor working group invitation),
(associated working group invitation)}
VIA
(SELECT *
FROM {(), (first)}
VIA EXTERNAL CONDITION)
POSITION OF OBJECT.*PERSON*
IN
{(national representative),
(sponsor working group member),
(associated working group member)}

OLD PRIORITY INVITATION:
{(SELECT *
FROM *IDENTIFY*
WHERE *NAME*= 'Priority',
invitation)}

5.7. General Commentary

To keep the size of this chapter manageable, several issues concerning the Working Conference Problem have not been addressed in context. Each is discussed here briefly in order to round-out the application of PAMS to the problem.

Not only, as suggested in the previous section, have many possible alternative definitions been omitted from the solution, so have many basic details. For instance, each *WORKING GROUP* should logically have attributes such as *NAME*, *NUMBER*, *CHAPTER*, and *MEETING PLAN* in addition to *MEMBER*. Nuances such as activities which "approve possible chairman" or "retract invitation" have not been included. Detailed pre-conditions to allow for exceptional conditions, such as missing referee reports, are ignored. What is presented, however, more than suggests how such things could be added to the solution.

Cardinality constraints were not considered in the solution presented — all bundle definitions were allowed to cover the full range of possible sizes. It is clearly appropriate to constrain some bundles such as *IFIP* and *TECHNICAL TOPIC* to single values. For others, including *PERSON.NAME* and *SESSION.CHAIRMAN*, the choice to constrain or not involves a design decision and thus possibly an alternative definition. Some constraints also involve dependencies. For instance, *GROUP PAPERS FOR SESSION* should be limited to some maximum number of papers *per session*.

An interesting situation arises in regard to the *ISSUE CALL FOR PAPERS* task. In the absence of any preventive measures, any instance of this task would issue *all CALL FOR PAPERS* instances in the database! Therefore a pre-condition must be added to select the appropriate instance of *CALL FOR PAPERS*. While it might make sense for the pre-condition to be based on an external condition, a more relevant condition would involve something having to do with the "current conference" or the state of a "conference selection" task. This situation applies throughout much of the IFIP Working Conference Problem. The necessary pre-condition must be added even in unexpected places, such as the constraints which cause *SPONSOR WORKING GROUP* and *ASSOCIATED WORKING*

GROUP to partition the specialization of *INVOLVED WORKING GROUP* on a conference by conference basis.

Lastly, virtually all the definitions presented in this chapter can be expressed in multiple, equivalent ways. The choice of notation for bundles, the inclusion/exclusion of extraneous names from the design, and the use of SQL-style syntax all contribute to this fact. In general, the approach of this chapter has favored clarity over necessity or conciseness, with some variations to illustrate particular features of PAMS.

CHAPTER 6

Contributions and Limitations

No work of this size and scope produces any lasting benefit unless a number of significant issues are addressed by means of retrospective analysis. Both the pros (contributions) and the cons (limitations) must be rationally weighed in such a way that the ultimate benefits (future directions) become reasonably apparent. This final chapter analyzes the previous five in such a light.

6.1. Contributions

On the plus side of the ledger, it is the author's contention that this work embodies several major contributions:

* *Categorization of database semantics*: The categorization and placement, in Chapter 2, of existing data modeling concepts along a scale of semantic content represents a major step forward in the analysis of semantic data models. Codd's notion that all models are *semantic* simply to more or less degree [Codd1979] now has an operational meaning: the relative placement of models along such a scale.

* *Defined classes of semantics*: Within the context of the above categorization, the notion that certain fixed points may be embodied by different styles of templates also represents a major advancement. Each template style represents a defined research target. Focus toward such specific goals can prevent the "hodge-podge" collection of modeling concepts into a modeling system which has no corresponding interpretation.

* *Identification of the premature generalization trap*: The notion, as discussed in Chapter 3, that some forms of semantic analysis introduce false regularities into application design has not been previously seen as such in the literature. The way in which several existing problems are related to the same fluke of human nature suggests that both the problems and their solutions are of lasting importance. The related nature of the problems also suggests a possible solution path: extending existing solutions to form solutions to unsolved variants of the trap.

* *Bundles as extended relations*: The introduction of the *bundle* as a modeling concept is perhaps the most widely applicable result of this work. Since the bundle is a well defined extension of Codd's *relation* to cover situations observed elsewhere in the literature, any modeling system based on the concept should prove immediately useful in simplifying at least some current research problems.

* *Value nomination*: The notion that individual values may automatically and transparently be substituted for bundles of values significantly simplifies the description of many database objects and operations. It provides a uniform mechanism for handling special cases which are otherwise ignored due to complication of the necessary data structures.

* *Approach to modeling database activity*: The Prototype Activity Modeling System presented in Chapter 4 provides a methodology for the description and use of dynamic

templates. PAMS is important not so much as a complete solution but rather as an "existence proof" that there are semantic data modeling systems which capture complex semantics yet are still based on interpretations, such as dynamic templates, which are simple for the user to understand.

* *Non-procedural thunks*: The embedding of non-procedural query specifications within the database itself is both new and novel. This concept allows for the dynamic definition of application entities as they are referenced and for the incorporation of process-related information within the database.

* *Activity monitoring*: By providing an automatic record of database activity (in particular, of aggregation construction and deletion) as a part of the database, a modeling system may free the database design from many "house-keeping" attributes which serve only to record the passage of events. Further, such monitoring can form the basis for a query language facility which supports the time-varying nature of relations.

* *Generalization through aggregation*: The notion that generalization (in its inverse form, specialization) can be reduced to an aggregation of objects is extremely significant since it reduces the number of orthogonal dimensions in semantic data modeling. Such reduction simplifies the interpretation associated with a model.

6.2. Limitations

As with virtually all research works, the results presented herein do suffer from some significant limitations. Those found here fall in two distinct categories: limitations which are imposed by the Prototype Activity Modeling System and limitations which are inherent in dynamic templates.

Within the scope of problems addressed by this work, perhaps the most profound limitation concerns the types of semantic dependencies addressed by the PAMS. Even though four types are considered (containment, feedback, operational, and state dependency), there are no doubt many additional types of dependencies which can account for apparent change within a dynamic template. This fact highlights the *prototype* nature of PAMS. As an example, consider one additional type:

* *Location* dependencies include any constraints on the logical position of one item with respect to another. For example, the sun sets somewhere to the west of where it rises.

It would appear that location dependencies are best modeled on a relative basis and that something akin to a state dependency clock could be used to extend PAMS. This does not imply that the solutions are easy, however, since location is a three-dimensional concept whereas time is one-dimensional. At best only the direction toward a "Non-Prototype" Activity Modeling System is clear.

Another basic limitation of PAMS is its reliance on pre- and post-conditions, and thus on SQL-style query specification extensions, to represent knowledge about operations. The decision to use SQL is based on the need to integrate all application and database objects and to use any object as an "operation". This, however, sacrifices some of the axiomatic advantages usually associated with pre- and post-conditions making any attempt to verify a PAMS application more complex.

As suggested above, the basic concept of a dynamic template also imposes limitations: dynamic templates are *defined* to include only a limited degree of semantics. It is quite

reasonable to define other types of templates which include lesser or further degrees. Some such possibilities include:

* *Perspective templates* where only conditionally defined alternatives are supported. This intermediate form between static and dynamic templates omits support for state dependency, thus obliterating the need for the *activity monitoring principle*. Implementation of a modeling system based on such templates would, therefore, be more immediately realizable than for PAMS.

* *Probabilistic templates* where a multivalued logic is used to express the probability of dependencies. In a dynamic template, the selection of alternative definitions is binary in nature; each decision is made strictly on the truth of what amount to assertions. Not all real world situations are so cut and dried. Sometimes there is insufficient information but there are well defined likelihoods with which the alternatives occur. For these situation it may be appropriate to associate a probability factor with each alternative of a conditional definition.

* *Fuzzy templates* where relationships are less exact in nature. As an extension of probabilistic templates, fuzzy templates might provide for situations where the definitions of the alternatives themselves are also imprecise. Many seemingly simple "facts" associated with abstract concepts are not simply stated. Even the color of an object, for instance, may require interpretation on the part of the viewer: where does "green" end and "blue" begin?

* *Simulation templates* where enough information about operations is included so that automatic simulation of the application may be done. Such templates cross the boundary between data models and simulation languages.

* *Organic templates* where inference is used to change the template. Thus the apparent change of definition supported by dynamic templates becomes actual change. One interesting approach here would be the integration of a PROLOG interpreter with a modeling system such as PAMS; this approximates the task of many "fifth generation" computer system research efforts.

6.3. Future Directions

Two possible future directions have already been suggested by previous sections: the implementation of a database management system based on PAMS and the theoretical extension of PAMS into areas such as organic templates. One could also further develop the environment in which PAMS is applied by turning PAMS into a complete information system design methodology. This might be accomplished by formalizing and refining some of the techniques used in Chapter 5 to solve the IFIP Working Conference Problem.

Perhaps the ultimate goal might be the integration of *all* these "future directions" into one unified whole. The target would be an integrated information system design *and* implementation methodology capable of inferential self-extension. Perhaps this is the definition of a "sixth-generation" software system.

References

[ANSI1975]
ANSI/X3/SPARC Study Group on Data Base Management Systems. Interim report. *FDT (ACM SIGMOD Bulletin) 7*, 2 (1975), 3-140.

[Anderson1982]
Anderson, T. Modeling time at the conceptual level. In *Proceedings of the Second International Conference on Improving Database Usability and Responsiveness* (June 1982), 273-297.

[Antonellis1981]
Antonellis, V. and Zonta, B. Modelling events in data base application design. In *Proceedings of the Seventh International Conference on Very Large Databases* (1981), 23-31.

[Aschim1982]
Aschim, F. and Mostue, B. IFIP WG 8.1 case solved using Sysdoc and Systemator. In *Information Systems Design Methodologies: A Comparative Review.* T. Olle, H. Sol, and A. Verrijn-Stuart (Eds.). North-Holland, Amsterdam, Netherlands (1982), v-vii.

[Bolour1983]
Bolour, A. and Dekeyser, J. Abstractions in temporal information. *Information Systems 8*, 1 (1983), 41-49.

[Bradley1978]
Bradley, J. Operations data bases. In *Proceedings of the Fourth International Conference on Very Large Databases* (1978), 164-176.

[Brodie1980a]
Brodie, M. and Zilles, S. (Eds.). Behaviour. In *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modeling* (June 1980), 53-61. Transcription of Workshop Session.

[Brodie1980b]
Brodie, M. and Zilles, S. (Eds.). What should be modelled? [*sic*]. In *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modeling* (June 1980), 40-42. Transcription of Workshop Session.

[CODASYL1971]
Data Base Task Group of the CODASYL Programming Language Committee. *Report.* Association for Computing Machinery, New York (April 1971).

[Chamberlin1974]
Chamberlin, D. and Boyce, R. SEQUEL: A structured English query language. In *Proceedings of the ACM Workshop on Data Description, Access, and Control* (1974), 249-264.

[Chamberlin1976]
Chamberlin, D., Astrahan, M., Eswaran, K., Griffiths, P., Lorie, R., Mehl, J., Reisner, P., and Wade, B. SEQUEL 2: A unified approach to data definition, manipulation, and

control. *IBM Journal of Research and Development 20*, 6 (November 1976), 560-575.

[Chen1976]
Chen, P. The entity-relationship model — Toward a unified view of data. *ACM Transactions on Database Systems 1*, 1 (March 1976), 9-36.

[Codd1970]
Codd, E. A relational model of data for large shared data banks. *Communications of the ACM 13*, 6 (June 1970), 377-387.

[Codd1979]
Codd, E. Extending the relational model to capture more meaning. *ACM Transactions on Database Systems 4*, 4 (December 1979), 397-434.

[Codd1980]
Codd, E. Data models in database management. In *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modeling* (June 1980), 112-114.

[Date1981]
Date, C. *An Introduction to Database Systems*. Addison-Wesley, Reading, Mass. (1981).

[Date1983]
Date, C. *An Introduction to Database Systems: Volume II*. Addison-Wesley, Reading, Mass. (1983).

[Ehrig1978]
Ehrig, H., Kreowski, H., and Weber, H. Algebraic schemes for data base systems. In *Proceedings of the Fourth International Conference on Very Large Databases* (1978), 427-440.

[Elliott1965]
Elliott, R. A model for a fact retrieval system. PhD Dissertation, Computation Center, University of Texas, Austin (May 1965).

[Foucaut1978]
Foucaut, O. and Rolland, C. Concepts for design of an information system conceptual schema and its utilization in the REMORA project. In *Proceedings of the Fourth International Conference on Very Large Databases* (1978), 342-350.

[Hammer1978]
Hammer, M. and McLeod, D. The semantic data model: A modeling mechanism for database applications. In *Proceedings of the 1978 SIGMOD Conference* (May 1978), 26-36.

[Hammer1981]
Hammer, M. and McLeod, D. Database description with SDM: A semantic data model. *ACM Transactions on Database Systems 6*, 3 (September 1981), 351-386.

[Ingerman1961]
Ingerman, P. Thunks: A way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM 4*, 1 (January 1961), 55-58.

[Kerschberg1976]
Kerschberg, L., Klug, A., and Tsichritzis, D. A taxonomy of data models. Technical

Report CSRG-70, Computer Systems Research Group, University of Toronto (May 1976).

[Lamersdorf1983]
Lamersdorf, W. Comparison of four relational data model specifications. *SIGMOD Record 13*, 3 (April 1983), 18-31.

[Leonard1981]
Leonard, M. and Luong, B. Information systems design approach integrating data and transactions. In *Proceedings of the Seventh International Conference on Very Large Databases* (1981), 235-245.

[Leveson1980]
Leveson, N. Applying behavioral abstraction to information system design and integrity. PhD Dissertation, Computer Science Department, University of California, Los Angeles (March 1980).

[Leveson1983]
Leveson, N., Wasserman, A., and Berry, D. BASIS: A behavioral approach to the specification of information systems. *Information Systems 8*, 1 (1983), 15-23.

[Lockemann1979]
Lockemann, P., Mayr, H., Weil, W., and Wohlleber, W. Database abstraction for database systems. *ACM Transactions on Database Systems 4*, 1 (March 1979), 60-75.

[MacDonald1982]
MacDonald, I. and Palmer, I. System development in a shared data environment: The D2S2 methodology. In *Information Systems Design Methodologies: A Comparative Review.* T. Olle, H. Sol, and A. Verrijn-Stuart (Eds.). North-Holland, Amsterdam, Netherlands (1982), 235-283.

[McGee1977]
McGee, W. The information management system IMS/VMS, part II: Data base facilities. *IBM Systems Journal 16*, 2 (1977), 96-122.

[McLeod1980]
McLeod, D. and Smith, J. Abstraction in databases. In *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modeling* (June 1980), 19-25.

[Mylopoulos1980]
Mylopoulos, J., Bernstein, P., and Wong, H. A language facility for designing database intensive applications. *ACM Transactions on Database Systems 5*, 2 (June 1980), 185-207.

[Olle1982a]
Olle, T. Comparative review of information systems design methodologies, stage 1: Taking stock. In *Information Systems Design Methodologies: A Comparative Review.* T. Olle, H. Sol, and A. Verrijn-Stuart (Eds.). North-Holland, Amsterdam, Netherlands (1982), 1-14.

[Olle1982b]
Olle, T., Sol, H., and Verrijn-Stuart, A. (Eds.). *Information Systems Design Methodologies: A Comparative Review.* North-Holland, Amsterdam, Netherlands (1982).

[Paolini1981]
Paolini, P. Abstract data types and data bases. PhD Dissertation, Computer Science Department, University of California, Los Angeles (1981).

[Parnas1972]
Parnas, D. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12 (December 1972), 1053-1058.

[Parnas1979]
Parnas, D. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering SE-5*, 2 (March 1979), 128-137.

[Rolland1979]
Rolland, C., Leifert, S., and Richard, C. Tools for information system dynamics management. In *Proceedings of the Fifth International Conference on Very Large Databases* (October 1979), 251-261.

[Schmid1975]
Schmid, H. and Swenson, J. On the semantics of the relational data model. In *Proceedings of the 1975 SIGMOD Conference* (May 1975), 211-223.

[Smith1977a]
Smith, J. and Smith, D. Database abstractions: Aggregation. *Communications of the ACM 20*, 6 (June 1977), 405-413.

[Smith1977b]
Smith, J. and Smith, D. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems 2*, 2 (June 1977), 105-133.

[Smith1980]
Smith, J. and Smith, D. A database approach to software specifications. In *Software Development Tools*. W. Riddle and R. Fairley (Eds.). Springer-Verlag, New York (1980), 176-204.

[Steel1975]
Steel, T. Data base standardization — A status report. In *Proceedings of the 1975 SIGMOD Conference* (May 1975), 65-78.

[Su1981]
Su, S., Lam, H., and Lo, D. Transformation of data traversals and operations in application programs to account for semantic changes of databases. *ACM Transactions on Database Systems 6*, 2 (June 1981), 255-294.

[Verheijen1982]
Verheijen, G. and van Bekkum, J. NIAM: An information analysis method. In *Information Systems Design Methodologies: A Comparative Review*. T. Olle, H. Sol, and A. Verrijn-Stuart (Eds.). North-Holland, Amsterdam, Netherlands (1982), 537-589.

[Verrijn-Stuart1982]
Verrijn-Stuart, A. CRIS: An introduction. In *Information Systems Design Methodologies: A Comparative Review*. T. Olle, H. Sol, and A. Verrijn-Stuart (Eds.). North-Holland, Amsterdam, Netherlands (1982), v-vii.

[Wasserman1979]
Wasserman, A. The data management facilities of PLAIN. In *Proceedings of the 1979*

SIGMOD Conference (May 1979), 60-70.

[Wasserman1980]
Wasserman, A. The design of PLAIN: Support for systematic programming. *Proceedings of the National Computer Conference 49* (May 1980), 731-740.

[Weber1978]
Weber, H. A software engineering view of data base systems. In *Proceedings of the Fourth International Conference on Very Large Databases* (1978), 36-51.

[Zloof1975]
Zloof, M. Query by example. *Proceedings of the National Computer Conference 44* (May 1975), 431-438.