Analysis of Methods

Final Report

Richard J. Mayer, ed.

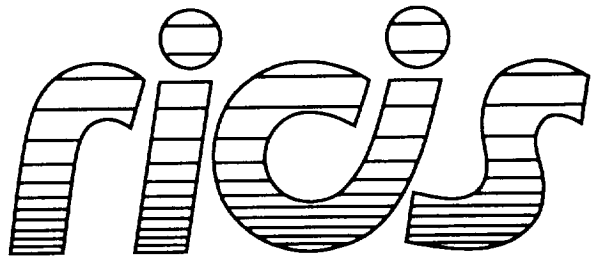
Knowledge Based Systems Laboratory
Texas A&M University

March 8, 1991

Cooperative Agreement NCC 9-16

Research Activity No. IM.06:
Methodologies for Integrated
Information Management Systems

NASA Johnson Space Center
Information Systems Directorate
Information Technology Division



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

TECHNICAL REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

Analysis of Methods

Final Report

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Dr. Richard J. Mayer, Keith A. Ackley, M. Sue Wells, Dr. Paula S.D. Mayer, Thomas M. Blinn, Louis P. Decker, Joel A. Toland, J. Wesley Crump, Dr. Christopher P. Menzel, Charles A. Bodenmiller and Michael T. Futrell of Texas A&M University; Stu Coleman and Timothy Ramey of PIM, Inc. and Dr. Tom Cullinane of Northeastern University. Dr. Peter C. Bishop served as RICIS research coordinator.

Funding has been provided by the Air Force Armstrong Laboratory, Logistics Research Division, Wright-Patterson Air Force Base via the Information Systems Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this research activity was Robert T. Savely of the Information Technology Division, NASA/JSC.

Analysis of Methods

KBSL - 89- 1001

Knowledge Based Systems Laboratory
Department of Industrial Engineering
Texas A&M University
College Station, TX 77843

Analysis of Methods

Edited by

Richard J. Mayer, PhD

Authors

Richard J. Mayer, PhD
Keith A. Ackley
M. Sue Wells
Paula S.D. Mayer, PhD
Thomas M. Blinn

Louis P. Decker
Joel A. Toland
J. Wesley Crump
Christopher Menzel, PhD
Charles A. Bodenmiller
Michael T. Futrell

Stu Coleman, PIM Inc.
Timothy Ramey, PIM Inc.
Tom Cullinane, PhD Northeastern Unversity

Final Report
March 8, 1991

Acknowledgements: This report describes ongoing research at the Knowledge Based Systems Laboratory of the Department of Industrial Engineering at Texas A&M University. Funding for the lab's research in Integrated Information System Development Methods and Tools has been provided by the United States Air Force Human Resources Laboratory, AFHRL/LRL, Wright Patterson Air Force Base, Ohio 45433, under the technical direction of USAF Captain Michael K. Painter, under subcontract through the NASA RICIS Program at the University of Houston.

Additional funding has been provided by Tandem Computer Corporation for broader coverage in the analysis of existing methods as represented by the Data Flow Analysis and Structure Chart chapters.

Table of Contents

1

Introduction

Information is one of an organization's most important assets. For this reason the development and maintenance of an integrated information system environment is one of the most important functions within a large organization. The Integrated Information Systems Evolution Environment (IISEE) project has as one of its primary goals a computerized solution to the difficulties involved in the development of integrated information systems. These difficulties involve such issues as:

- What activities are performed within the organization by either individuals or groups of individuals.
- What, how and when do these individuals or groups communicate.
- What information is required by these individuals or groups.
- How is this information to be presented to the individual users of the system.

To develop such an environment a thorough understanding of the enterprise's information needs and requirements is of paramount importance. This document is the current release of the research performed by the Integrated Development Support Environment (IDSE) Research Team in support of the IISEE project.

Our research indicates that an integral part of any information system environment would be multiple modeling methods to support the management of the organization's information. Automated tool support for these methods is necessary to facilitate their use in an integrated environment. An integrated environment makes it necessary to maintain an integrated database which contains the different kinds of models developed under the various methodologies. In addition, to speed the process of development of models, a procedure

or technique is needed to allow automatic translation from one methodology's representation to another while maintaining the integrity of both. The purpose for the analysis of the modeling methods included in this document is to examine these methods with the goal being to include them in an integrated development support environment. To accomplish this and to develop a method for allowing intra-methodology and inter-methodology model element reuse, a thorough understanding of multiple modeling methodologies is necessary.

Currently the IDSE Research Team is investigating the family of Integrated Computer Aided Manufacturing (ICAM) DEFinition (IDEF) languages $IDEF_0$, $IDEF_1$, and $IDEF_{1x}$, as well as ENALIM, Entity Relationship, Data Flow Diagrams, and Structure Charts, for inclusion in an integrated development support environment. The analysis of these methods began with the development of $IDEF_1$ metamodels for each method and a metamodel for the integrated database. This ongoing analysis has the following goals and should provide answers to many questions about the nature and application of system engineering methods..

- To gain a thorough understanding of the various methods.
- To determine where the methods overlap in order to assist in achieving information sharing between the methods.
- To gain the understanding necessary to translate manually from one method to another. The goal here will be to eventually provide automatic assistance in model translation.
- To begin to extract the theoretic foundations of each model method (if they exist).
- To develop the motivations (if they can be recovered) behind the development of the methods for the purpose of determining the original rationale for the development rather than how the methods have been applied.
- To understand how individual methods have been successfully applied, possibly outside of their original intent.
- To determine if an original engineering discipline exists for designing methods. This analysis represents reverse engineering on methodology development. It will assist the research team in determining what it takes to engineer a method.
- To determine which methodology should be used to discover information required or to answer questions encountered at each stage of the information system development process. This will involve determining what the application limits of each method

are and how the corresponding models or documents produced by a method can best be used.

- To determine what composes a good model of a given type. By "good model," we mean that the model is a syntactically and semantically correct model that concisely and correctly conveys the information intended by the author. Furthermore, a "good model" implies that the model was created using a methodology appropriate for the domain.

The process of creating metamodels for the various methodologies will allow the Research Team to define what information can be managed by a method in its native form. Knowing exactly what information is managed by two different methods is a precondition to the information integration of the methods and of automated model translation.

Traditionally, many people believe that many models contain the same information just packaged differently. However, in our work to date it is increasingly clear that this is not true. Little commonality between the information contents of models produced by different methods has been found. Different methods capture different aspects of the information system being designed. Furthermore, it is clear that a collection of methods, each managing its own part of the overall evolving system definition, is essential in the development of an integrated information system environment. All of the questions regarding these methodologies have not been answered. This document reflects our progress in the analysis of modeling methodologies and automated model translation.[1]

[1] For a description of theoretic formalizations that have been established for IDEF$_1$, $_{1X}$, $_3$, and information constraint specification languages, interested readers should also refer to Mayer R.J., et al. "Development Methodologies for Integrated Information Management Systems". Final Technical Report to United States Air Force Human Resources Laboratory. AFHRL/LRL, Wright Patterson Air Force Base,Ohio, Knowledge Based Systems Laboratory, Texas A&M University, 1988.; Menzel, C.P. and Mayer R.J., "IDEF$_3$ Technical Report", Knowledge Based Systems Laboratory Technical Report (KBSL-89-1006),1989.; Menzel, C.P. and Mayer R.J., "Theoretical Foundations for Information Representation and Constraint Specification", Knowledge Based Systems Laboratory Technical Report (KBSL-89-1001),1989.

2

$IDEF_1$: Information Modeling

Before attempting any of the other chapters in this report the Integrated Computed Aided Manufacturing (ICAM) DEFinition (IDEF) language $IDEF_1$ must be understood. $IDEF_1$ has a simple and clean syntax which can be learned quickly. On the other hand, there is an art to modeling in any methodology. $IDEF_1$'s design makes it imperative that the modeler understand proper modeling discipline.

As in each of the following chapters, this chapter will begin with a discussion of $IDEF_1$'s history and purpose and then move on to its syntax and semantics. Those familiar with the methodologies may not need to read the syntax and semantics sections, but keep in mind that many methodologies have several dialects. In order to understand the metamodels, it is important that the reader understand which dialect is being modeled. In general, the original definitions of methodologies are strictly followed.

2.1 History and Purpose

The family of IDEF methodologies is meant to provide methods and languages for discovery, representation, and consensus development of the views of an enterprise necessary to allow for planning and design of integrated information systems. That is, the IDEF methodologies were specifically developed for supporting the domain experts and systems analysts in gathering information about the existing environment and achieving consensus within the environment relative to those descriptions. $IDEF_0$ was developed to model the decisions, actions, and activities within a domain and the relationships among those activities. $IDEF_1$ provides the methods for discovery and representation of the logical structure and relations between basic information groups actually managed by an organization. $IDEF_2$

provides a method for development of quantitative simulation models that allow the study of time varying behavior of a system that is stochastic in nature. IDEF3 supports the direct capture of domain experts descriptions of process flow and object-state transitions. IDEF5 is under development to support the capture and representation of domain knowledge, concepts, and terminology (sometimes referred to as domain ontologies). IDEF1x was the first IDEF methodology to focus on support of system design activities. IDEF1x data incorporates criteria for efficient conceptual schema design. IDEF4 was developed later to support the design of object-oriented systems, particularly systems encompassing the use of object oriented databases. As a family, the IDEF methodologies provide the modeler with the ability to concentrate on views of an enterprise without using a "sledge hammer" methodology meant to model all views.

IDEF1 models the information managed within a system. It is closely related to but not a subset of IDEF1x. By providing a methodology for data modeling and consequently conceptual schema database design, the developers of IDEF1x added constructs which cloud the distinction between data which is kept about objects and the objects themselves. This was necessary since a conceptual schema by definition is a type of data dictionary (albeit a complex on-line dictionary used to provide both access and control to distributed electronic heterogeneous databases). Thus, a conceptual schema designer must develop a structure that can both contain the data objects and the information about those data object (such as their physical system location). IDEF1 however, was designed to be both more general and less committed to any particular implementation concept. In a properly developed IDEF1 model there should never be any misconceptions, only the information kept within an organization about objects (physical, abstract or data) is being modeled.

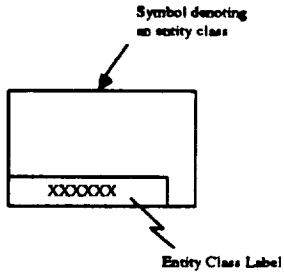
IDEF1 entities need not correspond directly to any particular object in the real world. The IDEF1 model represents the modeler's analysis results. The analysis method results in a reconstruction of the underlying structure and grouping of the information actually managed. In the real world these logical groups of attributes may be distributed over many data artifacts. Also, since data can be kept by the organization about any object (physical, abstract or data), this flexibility is necessary when attempting to establish information requirements. However, it is not constraining enough when doing database design (hence the need for IDEF1x, IDEF4, Entity Relationship (ER) and other design methods).

As with any of the IDEF methodologies, IDEF1 has primarily been used by defense contractors under contract to the Air Force. Hughes has a proprietary version of IDEF1 called ELKA (Entity Link Key

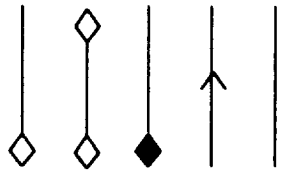
Attribute) [Ramey 85]. IDEF1's connection with defense projects is good in that a strong underlying analysis method has been developed for the application of IDEF1 modeling. With the emergence of the recognition of the need for a system development framework of methods and the availability of low-cost integrated tools for IDEF1 application, we can expect to see IDEF1 gain more widespread usage.

2.2 Syntax and Semantics

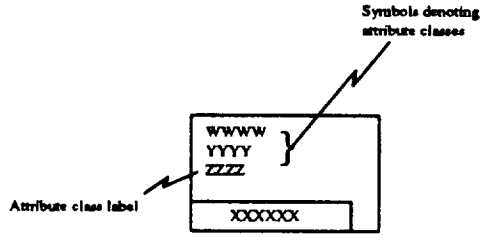
The lexicon of the IDEF1 language syntax consists of four basic symbols:



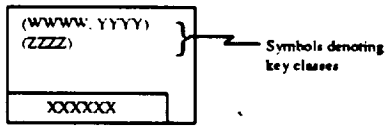
- Labeled boxes denoting entity classes,



- Labeled lines with five different types of diamond shaped terminators denoting relation classes,



- Labels inside the boxes denoting attribute classes,



- Parenthesized (or underlined) sets of labels denoting key classes.

2.2.1 Entity Class, Attribute Class, and Key Class

The concept of an "entity class" is meant to capture the notion of a basic information structure the extension of which at any point in time is a set of informational items called entities. The two basic concepts behind the notion of an entity are:

- they are persistent (i.e., the organization expends the resources (time, money, equipment or facilities) to observe, encode, record, organize and store the existence of individual entities).
- they can be individuated (i.e. they can be identified uniquely from other entities).

The IDEF1 language does not provide a means of representing individual entities. Only groups of entities which share exactly the same types of attributes can be represented. These groups from an IDEF1 view are called classes. A useful memory aid for this notion is to think of the entity class as a layout for a card file (see Figure 2.1). An entity class has a name and a unique identification number associated with it, along with a glossary entry and a list of synonyms. An entity class is represented by a rectangular box with the label of the entity class located in the lower left corner surrounded by a smaller rectangle and with the entity class number located in the lower right corner of the larger box.

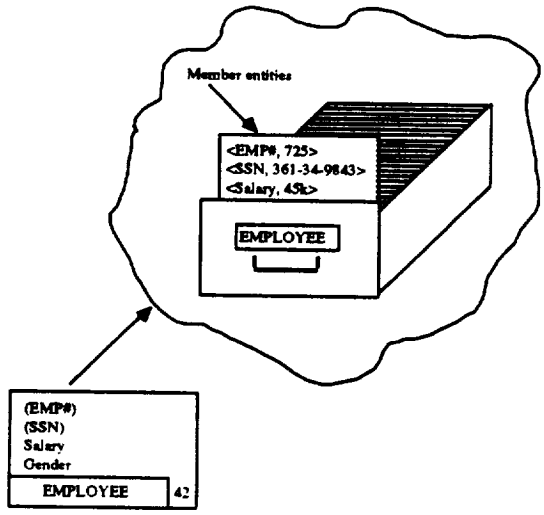
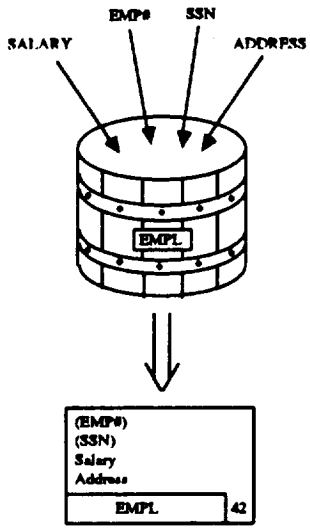


Figure 2.1 Card file interpretation of an IDEF1 entity class.

An entity class is actually defined by the set of attribute classes that define the characteristics of all the possible entities in all of its extensions. It is important to note that the set of attributes is more important that the notion conveyed by the label on the entity class



Bucket Analogy

name! In other words, one can think of the entity class as simply a labeled bucket with no meaning beyond that of the collection of attribute classes it contains (see insert for example). In fact, it is considered good practice to use an entity class label that does not name a physical or data object in the domain since that could confuse an uninformed reader. The labels of the attribute classes that define an entity class are simply listed in the entity class box below the key class designators and above the entity class label.

The occurrence of the same attribute class in multiple entity class definitions defines a relationship between those entity classes. In order to establish the existence dependency between such entity classes, one entity class must be determined to be the "owner" of the shared attribute class. Every attribute class that ends up being a part of an IDEF1 model has exactly one owner entity class. When deciding on the addition of an attribute class to an entity class; two rules must be followed. The first is referred to as the No-Null Rule. This rule states that no member of an entity class can take a null value for its attribute that corresponds to the added attribute class (Figure 2.2). The second rule, the No-Repeat rule, states that no member of an entity class can take more than one value at a time for its attribute that corresponds to the added attribute class (Figure 2.3).

Each entity class has associated with it at least one key class. A key class is just a special subset of the attribute classes which define the entity class. What makes such key class subsets special is that it can be determined that for any instance, the values of the attributes of that

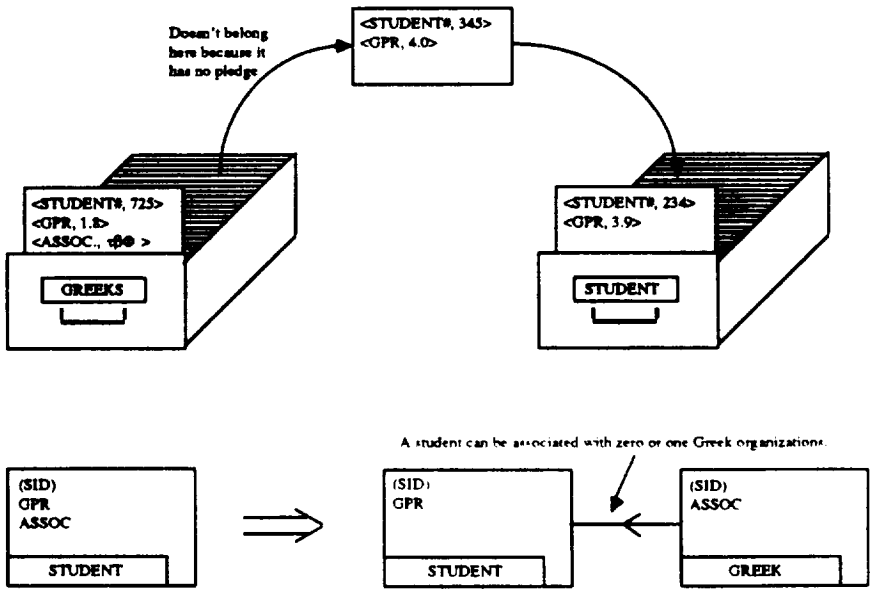


Figure 2.2 Example of the No Null Rule.

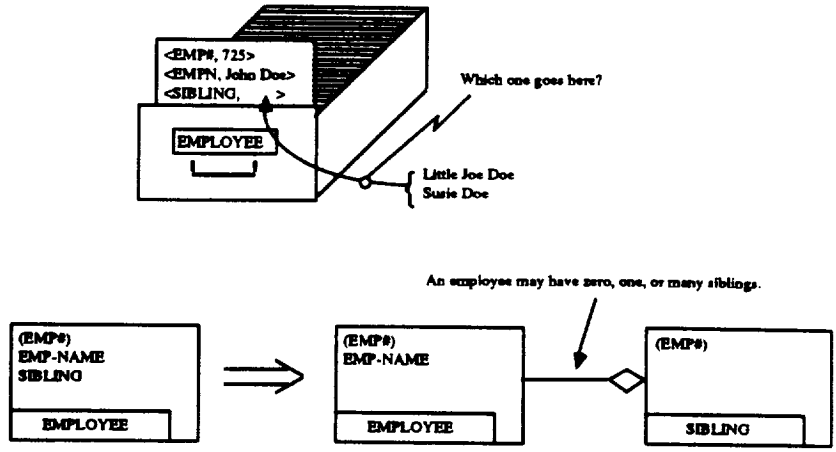
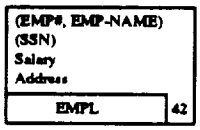


Figure 2.3 Example of the No-Repeat Rule.

instance (which correspond to the attribute classes in a key class), collectively, will uniquely identify that instance of the entity class from all other instances. In an IDEF1 diagram, the key class subsets are located in the upper left corner of the entity class for which the key class is being defined. Key classes are not named or labeled. A key class is denoted by enclosing the subset of attribute classes that make up the key class in parentheses or by underlining the subset. In the metamodels of this report we will use the parenthesis convention. It should be noted that entity classes are allowed to have multiple key classes. The multiple key classes would reflect multiple ways of identifying an entity class instance. For example, in a model of a typical business environment, an instance of an EMPL entity class might have multiple key classes. The first would consist of the employee's name in combination with an employee number. The second key class may consist only of the employee's Social Security Number. In both cases, an EMPL entity class instance could be uniquely identified by either key class (see insert for example).



2.2.2 Link (or Relation) Classes

A link is a binary relationship that exists between two entities. It is established by the sharing of a common attribute(s) which must assume the exact same value in each of the two entities involved in the link. In IDEF1 the generalization of all such links involving instances of the same two classes of entities and the same shared class(es) of attribute(s) is called a link class. A link class establishes a binary relationship between two entity classes that share a common attribute class. A link class is represented by a line running between the boxes of the two entity classes. A label, representing the name of the link class, is displayed over the line representing the link. Because

of the attribute class ownership property, a link indicates a dependence of one entity class on the other entity class. The dependent entity class is considered to be *existent dependent* since a member of that entity class cannot exist unless the corresponding member of the independent entity class exists. In general IDEF1 uses links to represent common types of organizational constraints (sometimes referred to as business rules) on the information that is managed. It should be noted that not all of the business rules can be represented with the standard IDEF1 language constructs. In a later section we will describe a constraint language called the Information Systems Constraint Language (ISyCL). ISyCL (pronounced "icicle") is used to augment the standard IDEF1 language as needed in this report to capture some of the more complex rules of individual methods.

A link class also has a cardinality associated with it, specifying the number of members of each entity class that can be involved in a relationship with a single member of the other entity class. Figure 2.4 shows the syntactic representation of a one-to-zero-or-one relationship. A link with this cardinality represents the fact that one member of the independent entity class can be associated with zero or one members of the dependent entity class. However, each member of the dependent entity class is associated with one and only one member of the independent entity class.

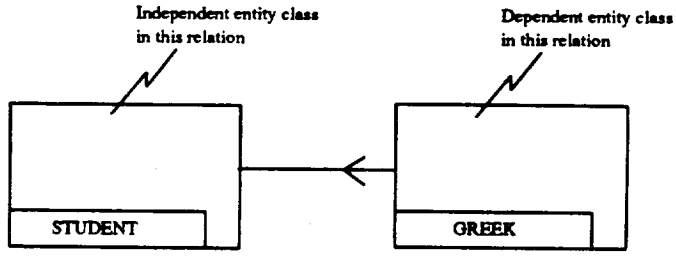


Figure 2.4 One-to-zero-or-one Link Class

Figure 2.5 shows the syntactic representation of a weak-one-to-many relationship. In this situation, an independent entity class member can be associated with zero, one, or many dependent entity class members. Again, each member of the dependent entity class is associated

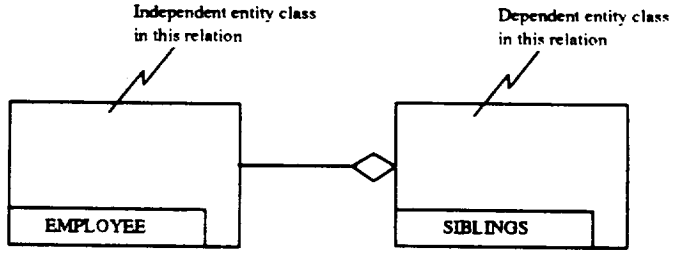


Figure 2.5 Weak-one-to-many Link Class

with one and only one member of the independent entity class.

Figure 2.6 shows the syntactic representation of a strong-one-to-many relationship. Here, the independent entity class member must be associated with at least one instance of the dependent entity class member. Again, each member of the dependent entity class is associated with one and only one member of the independent entity class.

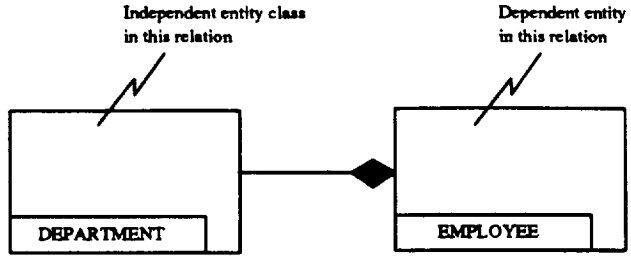


Figure 2.6 Strong-one-to-many Link Class

Notice that IDEF1 does not allow a many-to-many relationship or a zero-or-one-to-zero-or-one relationship in what is considered a final model. These relationships make the dependency situation ambiguous. The resolution of such uncertain situations (which often arise in the early phases of the corresponding analysis) often results in the analyst determination that the suspected relationship is unsupported by the analysis data. Alternatively the analyst may discover additional entity class(es) on which both of the entity classes involved in the "many to many" relationship are independent (an example of this is shown in Figure 2.7).

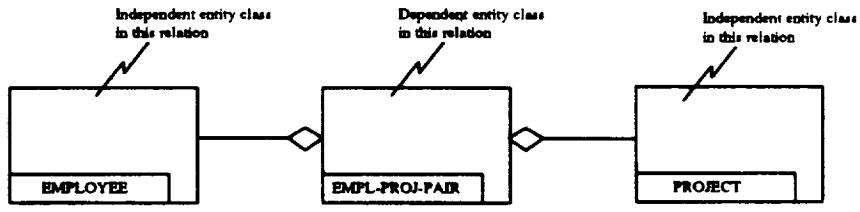


Figure 2.7 Resolution of a Many to Many Relation

Note also that, when specifying a one-to-many link class (either weak or strong), there is no way of constraining that link to a specific upper bound (for example, a one to five relationship). Such details are left to ISyCL if considered absolutely necessary.

2.2.3 Inheritance

Previously we noted that the sharing of attribute classes between two entity classes was the basis for declaring the existence of a link class between those entity classes. However, link classes are generally

suspected (or proposed) by the analyst prior to the discovery of exactly which attribute classes are shared. IDEF1 also places certain restrictions on which attribute classes may be (and must be) shared in order for a valid link class to be defined. When a link class is defined between two entity classes, certain information is shared between those entity classes. The attribute classes that make up the key classes of the independent entity class must become attribute classes for the dependent entity class. It is possible for the inherited attribute classes to become part of the key class of the dependent entity class. In fact, the attributes must become part of the key class when a link class has a one-to-zero-or-one link cardinality. In the case of a strong-one-to-many relationship the attributes that are shared cannot make up a key that would be a subset of the key of the independent entity class from which they came.

2.3 Metamodel

In this section the metamodel of IDEF1 (Figure 2.8) will be described in detail. Since the metamodels describe the information an information system would have to keep about a model, the IDEF1 method has been chosen for use on all of the metamodels.

One caveat about the link class labels used in the metamodels is that all link class labels have two parts. The first part of the label describes the relationship between the entity classes from independent to

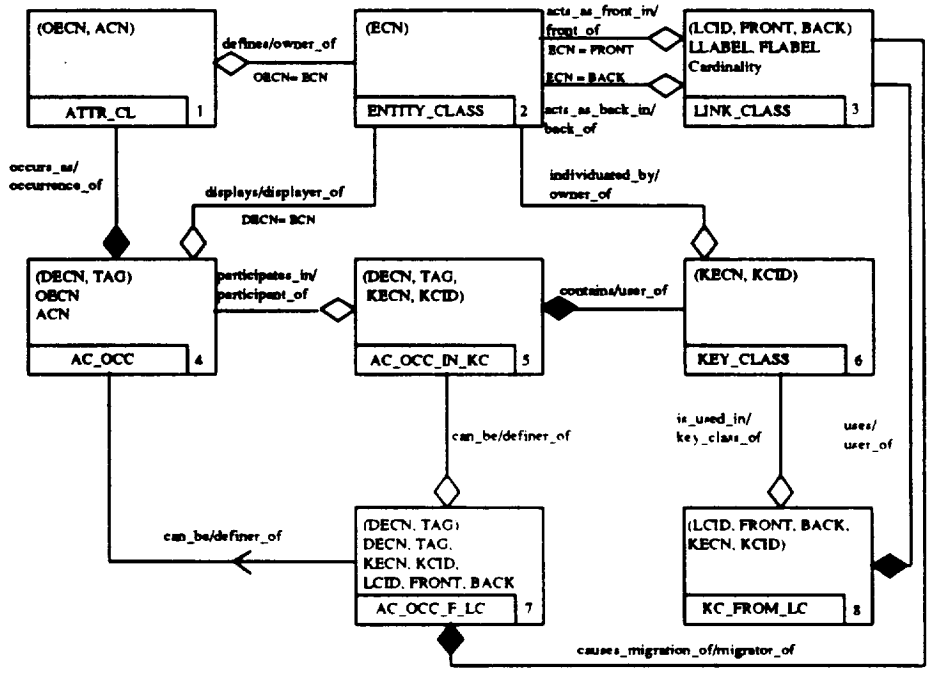


Figure 2.8 IDEF1 Metamodel of IDEF1

dependent and the second part describes the functional relationship between the dependent and the independent. In addition, some link class labels are augmented with "name1 = name2." This associates the inherited attribute class with the attribute class from which it was inherited (see Figure 2.8 for examples).

The metamodel of IDEF1 has been divided into four logical pieces to facilitate the explanatory process: 1) entity classes and attribute classes, 2) link classes, 3) key classes, and 4) attribute classes in key classes. Each piece will be described in the following sections.

2.3.1 Entity classes and Owned Attribute Classes

The key class of the entity class ENTITY_CLASS is made up of the single owned attribute class called the Entity Class Name (ECN) (see Figure 2.9). An entity class defines zero, one, or many attribute classes. An example of the need for a weak-one-to-many relationship is that an entity class serving as a dependent entity class in a one-to-zero-or-one relationship may not define an owned attribute class.

The entity class ATTribute CLass (ATTR_CL) is uniquely identified by its Attribute Class Name (OAN) and the Owner Entity Class Name (OECN) which defines it. Each entity class is made up af a set of attribute classes which define the properties for the entity class. This relationship is defined by the one to zero, one, or many link between ENTITY_CLASS and Attribute Class OCCurrence (AC_OOC). The entity class AC_OCC is uniquely defined by the key class consisting of two attribute classes: 1) the inherited attribute class Displaying Entity Class Name (DECN) and 2) the owned attribute class TAG which represents the label of the attribute class that is to be displayed

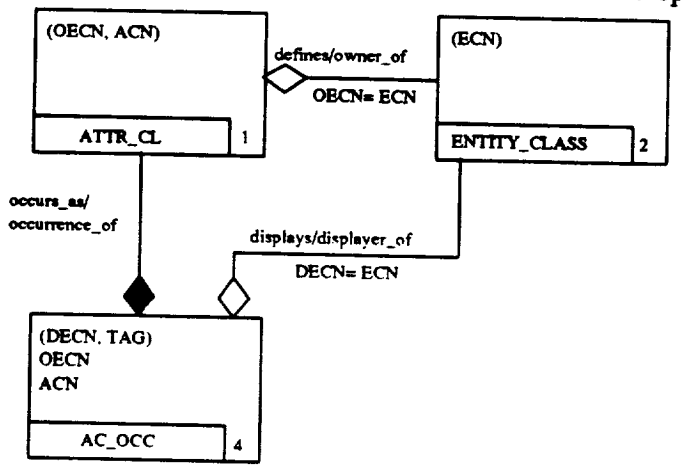


Figure 2.9 Metamodel of Entity and Attribute Classes.

in the entity class represented by DECN. The entity class AC_OCC also contains two inherited non-key attribute classes: OECN and ACN. These inherited attribute classes contain the information necessary to determine which attribute class this class is an occurrence of.

2.3.2 Link Classes

A link class represents a binary relationship between two entity classes as shown by the metamodel of link classes in Figure 2.10. A link class is uniquely identified by a key made up of three attribute classes: 1) a Link Class IDentifier (LCID), 2) the independent entity class participating in the link class (FRONT), and 3) the dependent entity class participating in the link class (BACK). However, a link constraint exists that states that every entity class must participate in at least one link class. This is represented by the following ISyCL constraint.

```
for_all e of entity_class ENTITY_CLASS
    (for_some l of entity_class LINK_CLASS
        (e = front_of(l)
        or
        e= back_of(l)))
```

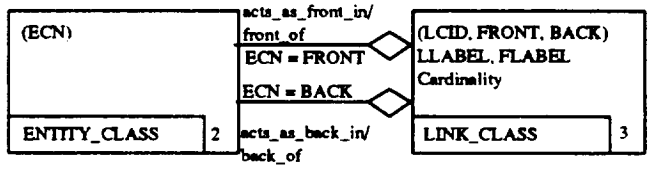


Figure 2.10 Metamodel of Link Classes

The entity class LINK_CLASS contains three additional attribute classes: 1) the owned attribute class Link LABEL (LLABEL) which is the label on the link representing the relationship from the independent entity class to the dependent entity class, 2) the owned attribute class Functional LABEL (FLABEL) representing the functional relationship between the dependent and the independent, and 3) an owned attribute class CARDINALITY which keeps track of the cardinality of the link class. The attribute values that are possible for the attribute class CARDINALITY include one-to-one, weak-one-to-many, and strong-one-to-many.

2.3.3 Key Classes

Every entity class ENTITY_CLASS has at least one entity class KEY_CLASS associated with it. As shown in Figure 2.11, the entity class KEY_CLASS is identified by the owned attribute class Key Class IDentifier (KCID) and the inherited attribute class Key Entity Class Name (KECN). The inherited attribute class contains the information necessary to determine the owner entity class for a given key class.

In addition, the model must keep track of the information needed to show which key classes migrated through which link class. This information is modeled using the entity class Key Class FROM Link Class (KC-FROM-LC). The key class of the entity class KC-FROM-LC is made up of both the key class from the entity class KEY_CLASS and the key of the entity class LINK_CLASS.

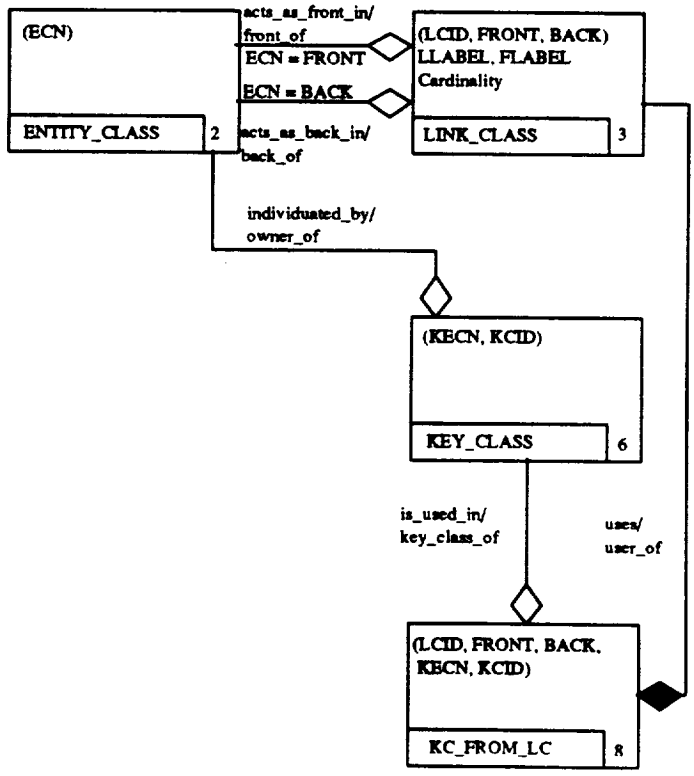


Figure 2.11 Metamodel of Key Classes

2.3.4 Attribute Classes in Key Classes

Key classes are made up of a collection of attribute classes. However since an attribute class can participate in multiple key classes of a given attribute class, it was necessary to add the entity class Attribute Class OCCurrence IN Key Class (AC_OCC_IN_KC) as shown in

Figure 2.12. The key class for the entity class AC_OCC_IN_KC is the union of the key classes for the entity classes AC_OCC and KEY_CLASS for which the entity class AC_OCC_IN_KC is dependent. In addition, since the attribute classes contained in a key class can be made up of either owned or inherited attribute class, another entity class Attribute Class OCCurrence From Link Class (AC_OCC_F_LC) had to be added to the model to record the fact that a given attribute class migrated across a link class.

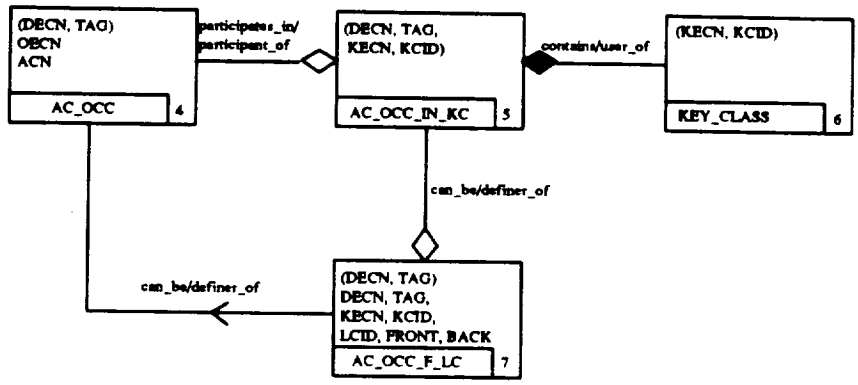


Figure 2.12 Metamodel of Attribute Classes in Key Classes

2.4 Strengths and Weaknesses

One of the weaknesses of the IDEF1 methodology is the fact that a modeler cannot talk about attribute values. That is, the methodology does not allow the modeler to talk about instances of an entity class and the values of the attributes of an instance of an entity class. Only the entity class as a whole can be discussed.

2.5 Tips and Traps

The IDEF1 methodology is an iterative development process which is observation based and contains the five following distinct phases:

- Phase 0 — context setting and data collection
- Phase 1 — entity class definition
- Phase 2 — link class definition
- Phase 3 — key attribute class definition
- Phase 4 — non-key attribute class definition

In phase 0, the model's context, viewpoint, and purpose are established. The context describes the subject and the boundary of the model. The perspective from which to interpret and understand the

model is defined by the viewpoint. The intent and objectives of the model are defined by the model's purpose. In addition, the collection and organization of data about the domain is started. In an IDEF1 model neither an entity class nor an attribute class may be introduced unless it can be traced back to some data which is either:

- currently managed in the domain, or
- a requirement to be managed in the future in the domain.

The model diagram begins taking form in the entity class definition phase. In this phase, the modeler defines the candidate entity classes of the model.

Similarly, in the link class definition phase, the relations between pairs of entity classes are defined. These relations are inferred from the entity class definition of the previous phase.

Next comes the definition of the key attribute classes, that is, those attribute classes that are needed to define the key classes of an entity class are defined in this phase along with the key classes.

In the last phase, all non-key owned attribute classes are defined for all of the entity classes.

2.6 Integration With Other Methodologies

Based on the metamodels of each of the methodologies, the IDSE Research Team is seeking to enable automated model translation. This allows information represented in one model to be translated into the equivalent representation in another methodology if one exists. Consequently, the modeler is able to use multiple methodologies without having to repeatedly enter equivalent information in each methodology.

2.7 Conclusions

A brief description of the IDEF1 method has been included to help familiarize and refresh the reader's knowledge about the method in order for that he or she may better understand the metamodels presented in this document. This chapter does not attempt to be an authoritative description of the IDEF1 method; it provides only a brief and concise description of IDEF1.

The metamodel of IDEF1 presented serves as an integration platform that the IDSE Research Team will use in the pursuit of a neutral information representation schema.

Appendix A. Abbreviations used in the IDEF1 Metamodel

AC_OCC: Attribute Class OCCurrence; an entity class.

AC_OCC_F_LC: Attribute Class OCCurrence from Link Class; an entity class.

AC_OCC_IN_KC: Attribute Class OCCurrence in Key Class; an entity class.

ACN: Attribute Class Name; part of a key class of ATTR_CL.

ATTR_CL: ATTRibute CLass; an entity class.

BACK: synonym for ECN.

DECN: Displayer Entity Class Name; part of the key class of AC_OCC.

ECN: Entity Class Name; occurs in the first key class of ENTITY_CLASS, uniquely identifies an entity class.

ENTITY_CLASS: an entity class.

FRONT: synonym for ECN.

KC_FROM_LC: Key Class from Link Class; an entity class.

KCID: Key Class IDentifier; part of the key class of KEY_CLASS.

KECN: Key Entity Class Name; part of the key class of KEY_CLASS.

KEY _CLASS: an entity class.

LCID: Link Class IDentifier; occurs in the key class of LINK_CLASS, uniquely identifies a link class.

LINK _CLASS: an entity class.

OECN: Owner Entity Class Name; part of a key class for ATTR_CL.

TAG: part of the key class to AC_OCC.

References

Mayer, R. J., IDEF1 - Information Modeling; Theory and Practice, Department of Industrial Engineering, Texas A&M University, 1988.

Ramey, T. L., Entity Link Key Attribute Semantic Information Modeling, Internal Technical Report, Hughes Grounds System Group, Fullerton, CA, October 24, 1985

Menzel, C., and Mayer, R., J., Theoretical Foundations for Information Representation and Constraint Specification, Technical Report, Knowledge Based Systems Laboratory, Texas A&M University, March 6, 1991.

3

IDEF0: Method for Function Modeling

This chapter introduces the history and purpose of the Integrated Computer-Aided Manufacturing (ICAM) DEFinition (IDEF) language IDEF0 method. Next, it briefly introduces the syntax and semantics of the IDEF0 method. Finally (and most important) the paper describes a complete IDEF1 metamodel of IDEF0. The purpose of the metamodel is to act as an integration platform with other methodologies such as IDEF1, IDEF1x, IDEF3, IDEF4, ENALIM, ER, Data Flow, and Structure Charts.

3.1 History and Purpose

The IDEF0 technique is based entirely upon a cell modeling technique known as the Structured Analysis and Design Technique (SADT) [Ross 81]. The Air Force Computer Aided Manufacturing (AFCAM) program in 1973 developed the foundations of the method through a joint effort with Boeing, and Softech [Buffum 74]. The method was based on the principles of "Human directed Activity Cell Modeling" of Dr. Shizuo Hori. Dr. Ross combined these basic principles with concepts that had evolved from his pioneering work in software engineering and programming language design to form in a structured technique for system analysis and a language for effective communication of the analysis results.

The purpose and philosophy of the resulting method is best stated in the original development report [Buffum 74] as follows:

> Structured analysis is founded on very simple basic principles that stem from the primary contention that: *To divide is to conquer, providing that it is clear how the divided pieces are structured together to constitute the whole.*

> Repeated application of this principle, with suitably simple notation, makes it possible to cover any subject from any point of view to any desired degree of completeness. The primary discipline quite simply is that "Everything worth saying about anything worth saying something about can be said by talking about six or fewer pieces." A true believer in these observations will automatically appreciate structured analysis but reduction to practice of the full discipline is a significant challenge. The primary objective of structured analysis description is to communicate completely and effectively. It must be clear just what is being said, and what is meant by what is being said. As long as clarity is achieved, then both agreement and disagreement can be accommodated.

The AFCAM program used this technique to build the first functional architecture of aerospace manufacturing. Following the AFCAM application Softech continued to evolve the resulting method into a software design technique. In 1976 the Air Force Integrated Computer-Aided Manufacturing (ICAM) follow on program employed the commercial version (known as SADT) to build a composite architecture of manufacturing as the first step in planning the ICAM program. In 1978 Doug Ross, Clair Feldman, and Richard Mayer took on the task of reworking the SADT method, cutting out the design principles and specializing the method to be a technique for:

- enhancing communication among the domain experts.
- performing non-departmentalized functional analysis of large organizational systems for information integration planning (a' la Dr. Joe Harrington).
- organizing the thought process of planners and analysts.

The IDEF0 method is used for modeling the functions of an organization (decisions, actions, and activities) and the relationships between those functions. Since the IDEF0 method and syntax incorporates (for function modeling) many of the early concepts of structured programming and design, the method supports the following principles [Ross 75]:

- the Modularity Principle: break the problem analysis results into its component parts and formalize the relations (protocol of interface) between those parts;
- the Abstraction Principle, identify common properties of functions and objects and define new functions or objects which can stand for classes defined by the common properties.

- the Hiding Principle: display only the level of detail that is relevant to the aspect of the model being viewed
- the Localization Principle: group activities or objects together that function to solve a particular problem.

The graphical language of the IDEF0 method supports these principles by allowing the author of a model to represent his results in terms of activity descriptions and the objects which form the relations between those activities arranged in an hierarchical structure. The root of this structure summarizes the results at the most abstract/general level. Each of the lower level nodes in the final tree structure provides more specific information than its parent. As the hierarchy is traversed downward the tree expands, unfolding the details of both the activities and the objects which form the relations between the activities. Figure 3.1 illustrates this hierarchical structure aspect of the syntax of the IDEF0 modeling method.

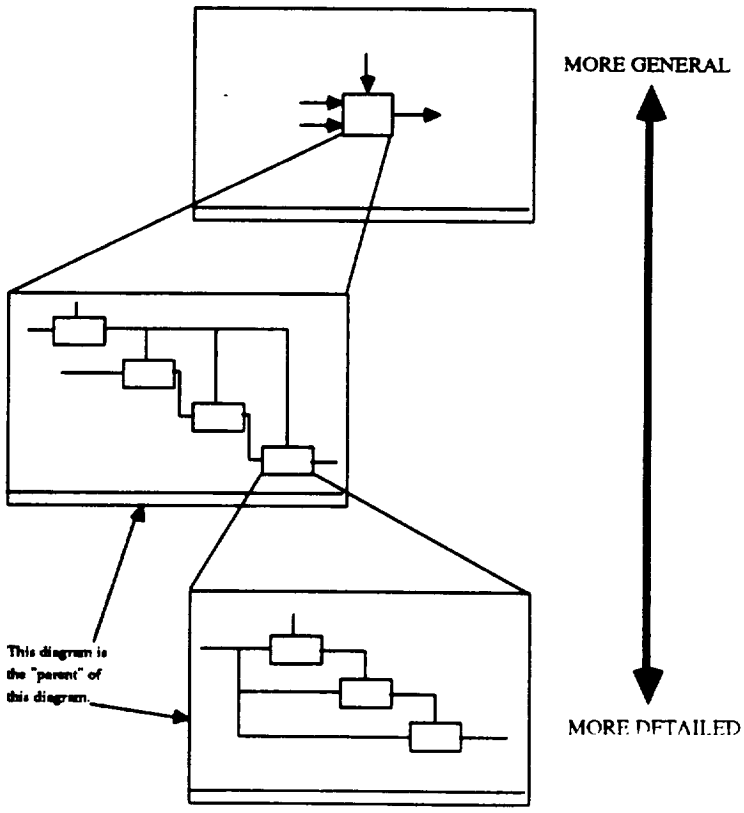


Figure 3.1 Example IDEF0 Decomposition

3.2 Syntax and Semantics

The syntax and semantics of the IDEF0 method evolved over many years through extensive human factors evaluations largely conducted by Doug Ross. The result is a language that, if used correctly, has proven capable of expressing functional architectures that are easy to understand. The following sections describe the basics of the language syntax and the "common sense" notion of the use semantics of this language.

3.2.1 Basic Symbols (IDEF0 lexicon)

The fundamental building blocks of the IDEF0 method language are labeled boxes denoting classes of functions (decisions, actions, or activities) and labeled arrows denoting the conceptual or real objects that form the relations or interfaces between the activities (Figure 3.2).

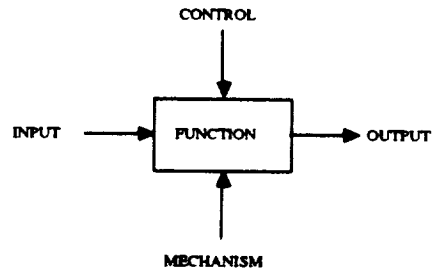


Figure 3.2 Generalized Function or Activity Box

Two types of diagrams are supported in the IDEF0 language "Context" diagrams and "Decomposition" diagrams. A context diagram displays a single activity box with its associated concepts (see Figure 3.3). A decomposition diagram displays three to six activity boxes each with their associated concepts. The decomposition diagram also displays the relations between activities formed out of the shared concepts between the activities denoted by arrows from one activity box to another (see Figure 3.4). An IDEF0 model is defined

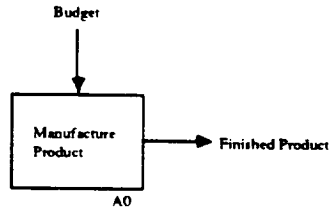


Figure 3.3 Example of a context diagram.

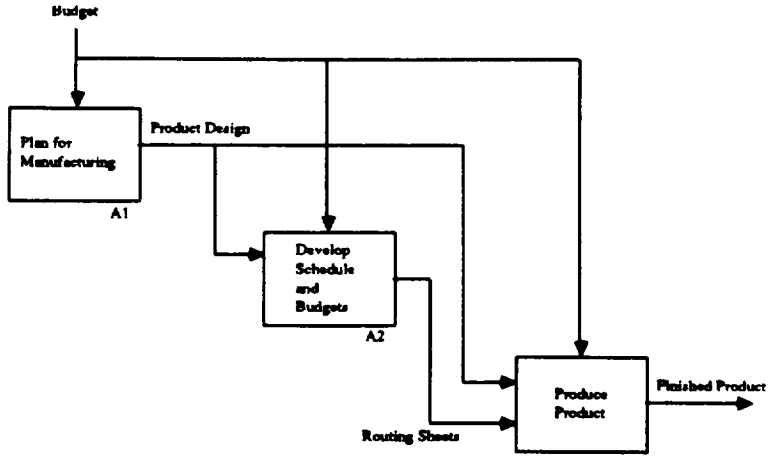


Figure 3.4 Example of a decomposition diagram.

as a context diagram and a set of decomposition diagrams along with a set of glossary data sheets (one for each IDEF0 model element). From this simple graphical lexicon and the following set of grammar rules, a model can be developed that is both concise and easily understood.

3.2.2 Grammar Rules for Function Descriptions

Figure 3.2 illustrates the basic structure of the representation of a function or activity in IDEF0. The position of the arrows entering and leaving the box represent the classification of the role a concept plays in its association with an activity. The four roles are input, control, output, and mechanism (ICOMs). The inputs enter the box from the left. They represent the concepts that are transformed in the execution of the function. The concepts serving in the control role enter at the top of the box. Concepts used as controls are assumed to influence how the function is performed. Concepts representing mechanisms are represented by arrows that attach to the box from the bottom. They represent the means by which the function is accomplished. For example, trains might be a mechanism of the activity "ship goods". The concepts which are represented by arrows that exit the box from the right represent the results produced by the function.

The IDEF0 language grammar requires that each function have at least one control and one output to be valid. There is no hard limit on the number of inputs, controls, outputs, and mechanisms that can be connected with a function, but good practice limitations are four to six of each. More than four to six is difficult to read and cannot be drawn legibly by hand or computer (without reduction). Remember that IDEF0 models are not intended to be specifications but rather vehicles for enhancing communication. If they are made unreadable by unnecessary clutter then they are generally useless.

Information about each function in an IDEF0 model can come from seven sources:

- the connotations of the name of the function.
- the position of the function at a level in the hierarchy.
- the glossary associated with the function.
- the concepts associated with the function.
- the parent of the function.
- the relationships of a function to its siblings on a diagram.
- the decomposition of a function into its children.

The IDEF0 language provides special syntactic elements for each of these sources of information. The name of the activity in the box covers the first. The position of the activity in the hierarchy is encoded in a unique number associated with each activity box. Each node number is prefixed with the capital letter "A". The root node is numbered with a 0. All the rest of the nodes are numbered with the number of their parent followed by a number representing their relative position with their siblings (see Figure 3.5). A textual glossary entry is associated with each activity (and concept). The concepts associated with an activity can be determined directly from the diagram. The source/sink of those concepts if local to the diagram can be traced on that diagram. If the source/sink is from the parent diagram then a code (called an ICOM code) provides the documentation for traceability to the parent level. Thus it is the physical arrows and ICOM codes that allow the communication of information relative to the relationship between individual (or groups of) activities. The description of an activity is not actually considered to be captured in the text but rather in the decomposition diagram associated with that activity. Every activity can be decomposed into three to six functions. This range was chosen (again for human factors considerations) to prevent a function from being described in too much or too little detail. Each time a decomposition occurs it is supposed to contain a detailed description of the parent function. Starting at the

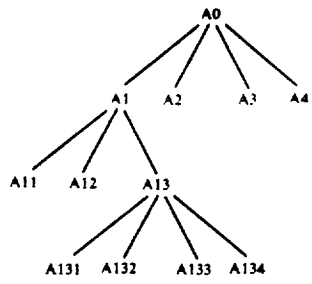


Figure 3.5 Example of the node numbering schema.

top, the process is recursive with each new level of decomposition giving more detail about an activity is to better describe the processes that occur. Again, Figure 3.1 illustrates the idea of decomposition into greater levels of detail. This characteristic of IDEF0 is consistent with hierarchical, top-down design approaches using refinement techniques.

3.2.3 Concepts

A concept is a piece of information, knowledge, data or physical object that is produced and/or consumed by an activity in an IDEF0 model. The term concept is used to include both tangible and intangible items. That is, concepts can be either actual things (e.g., documents and machined parts) or abstract ideas (e.g., production capacity, experience, problems, or sales quotas). This allows IDEF0 to model enterprises in many different domains. A key capability of the IDEF0 method and language is its support for the representation of the internal structure of these relation forming concepts. Concepts can divide and combine to form other concepts. A concept can split into two copies or spread into two different concepts. Also, two copies of a concept can join into a single copy or two different concepts can merge into a single concept. This capability of IDEF0 allows complex relationships between activities to be represented. Figure 3.6 shows an example of concepts spreading, splitting, joining, and merging.

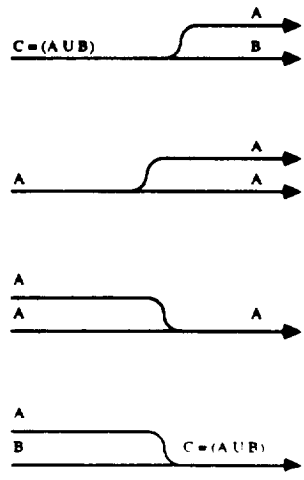


Figure 3.6 ICOM Spread, Split, Joint, and Merges

3.3 Metamodel

The following is a discussion of the IDEF1 metamodel of IDEF0. The metamodel is intended to capture the information managed in an IDEF0 model. Syntactic structures which can be directly generated from this information are not included in the metamodel. Therefore, for example, there will be no references to arrows in the metamodel since they can be reconstructed knowing the relations between the activities, and what concepts form these relations. Similarly the derivable information such as ICOM codes and activity numbers do not appear in the metamodel. These are artifacts of the diagram and not part of the information that is modeled. Also, the metamodel does not attempt to model real world processes described by an instance of an IDEF0 model. It models only the information managed in the method.

The IDEF1 metamodel of IDEF0 as shown in Figure 3.7 will be divided into four logical units to facilitate the discussion of the

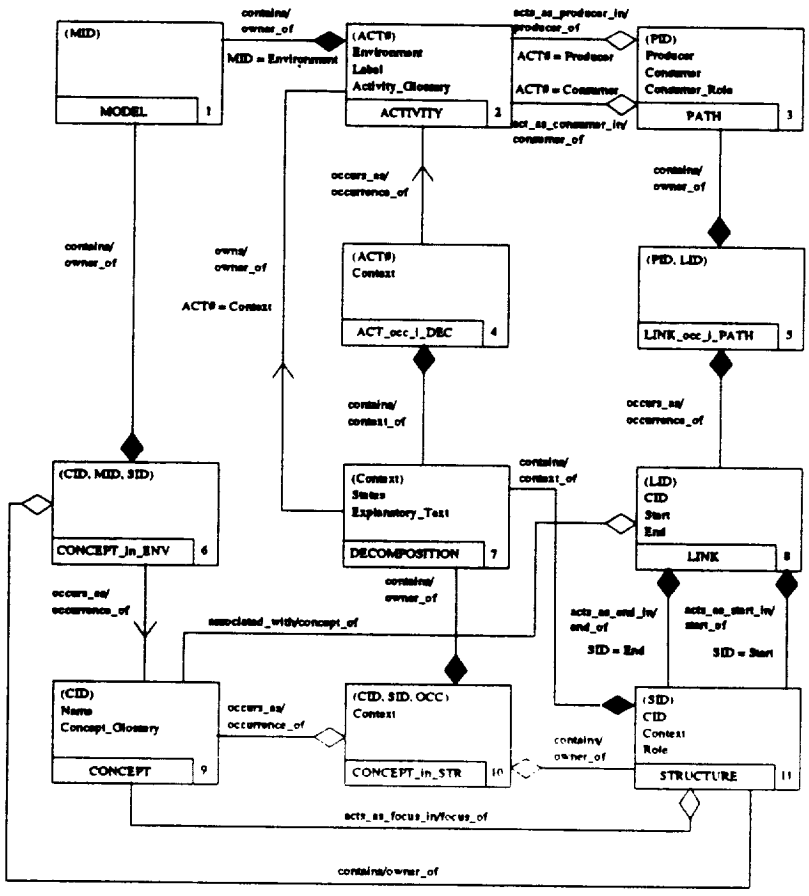


Figure 3.7 IDEF1 Metamodel of IDEF0

metamodel. The first logical unit will cover activities and the decomposition of activities. Next, the idea of structures and concepts will be introduced. Next, links will be presented. Finally, the idea of links and paths will be discussed.

3.3.1 Activities and Decompositions

The metamodel portion for activities and decomposition is shown in Figure 3.8. In IDEF0 an activity carries the information relative to the environment (the model at the most abstract level) in which it belongs. Each activity may or may not have a decomposition as represented by the one-to-zero-or-one link class from the ACTIVITY entity class to the DECOMPOSITION entity class. The key class of the parent activity of the decomposition serves as the key class of the DECOMPOSITION entity class.

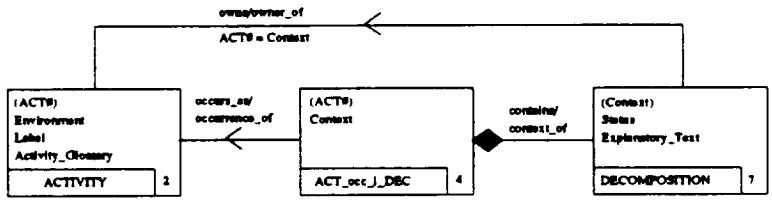


Figure 3.8 Metamodel of Activities

Each activity may participate in one and only one decomposition. This is because, by definition, an IDEF0 model is a rooted acyclic tree with the root activity not occuring in any decompositions. Thus, the one-to-zero-or-one link class from the ACTIVITY entity class to the ACTIVITY_occ_i_DEC (activity occurrence in decomposition) entity class represents the fact that an activity may or may not be contained in a decomposition.

The strong one-to-many link class from the DECOMPOSITION entity class to the ACTIVITY_occ_i_DEC entity class represents the fact that if an activity has a decomposition then the decomposition of the activity must contain between three and six activities. This constraint is represented by the following ISyCL statement:

```
for_all d of entity_class DECOMPOSITION
    (3 <= (length (contains(d)))) and ((length (contains(d))) <= 6)
```

3.3.2 Structures and Concepts

In trying to model the spreads, splits, joins,and bundles in IDEF0, a new construct called a structure was introduced into the model. The small numbered squares in Figure 3.9 represent structures. Every spread, split, join, and merge occurs at a structure. Structures are

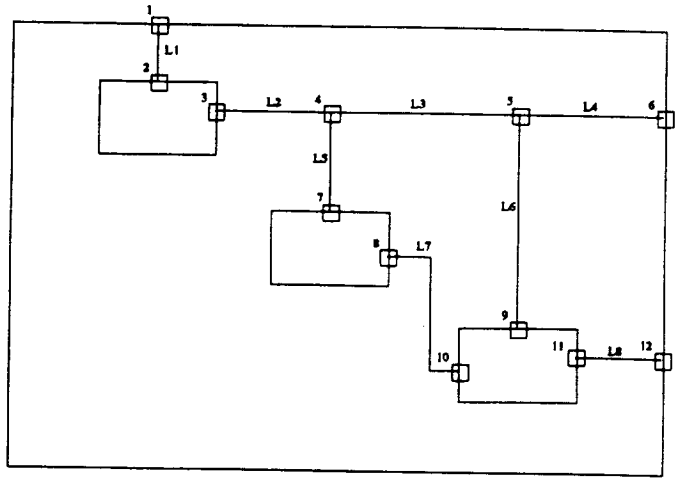


Figure 3.9 An example of structures

connected by links which, when chained, form a path between activities. Structures also occur at boundaries of a decomposition diagram to model flows into and out of the decomposition. Thus, all paths begin and end at a structure that is located at an activity or on the boundary of a decomposition.

Concepts are modeled by the CONCEPT entity class as shown in Figure 3.10. The CONCEPT entity class has an owned attribute class

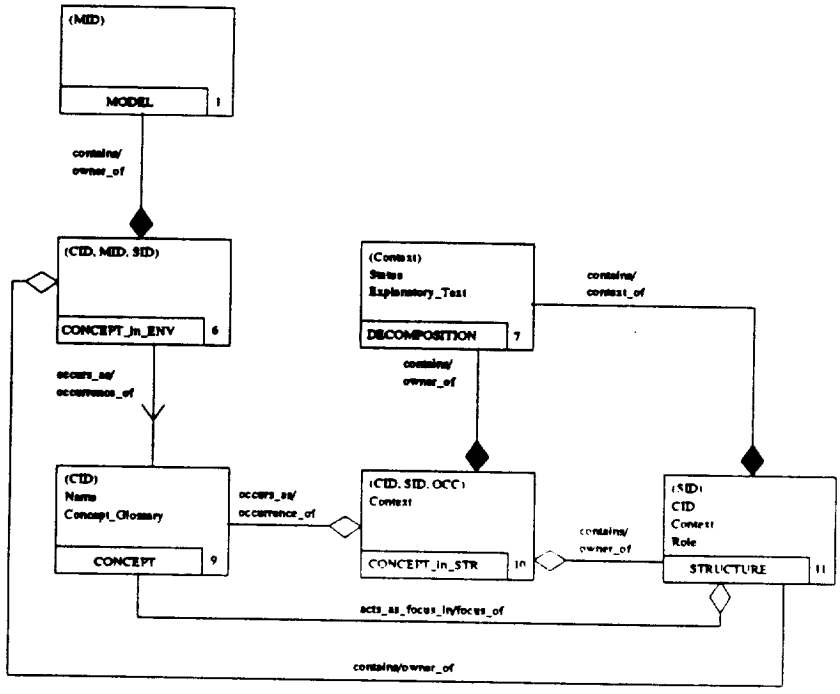


Figure 3.10 Metamodel of Concepts and Structures

NAME and a unique concept identifier (CID) symbol acting as the key class for the CONCEPT entity class. In addition, a CONCEPT entity class has an owned attribute class CONCEPT_GLOSSARY.

A concept can participate as the defining or focussing concept of zero, one, or many structures. The defining concept is the primary concept associated with a structure in a spread, split, join, or bundle. The CONCEPT entity class can be associated with zero, one, or many occurrences of the STRUCTURE entity class.

A STRUCTURE entity class also contains information about the context (decomposition) where the structure is located and a unique key identifying the structure (SID). In addition, a STRUCTURE entity class contains the owned attribute class ROLE. This attribute class specifies whether the structure is serving as a spread, split, join, or merge. If the structure is used as a spread, split, or pass-through (e.g., as in tunneling), the defining concept is the concept entering the structure. If the structure is used as a join or merge then the defining concept is the concept leaving the structure.

The concepts defined by the focus concept of a structure are modeled by the CONCEPT_in_STR (concept in structure) entity class. Since the same concept may exit or enter a structure multiple times, the key class of the CONCEPT_in_STR contains the focus concept identifier, the structure identifier, and a unique occurrence number. For example, a "distribute product" activity may produce an output that links to two activities. The output "product" may be used as input by both a "market product" activity and a "use product internally" activity.

The final entity class CONCEPT_in_ENV (concept in environment) in Figure 3.10 is used to model tunneled concepts. A tunneled concept is one that does not exist on the parent or child (decomposition) diagram of the current decomposition. That is, the concept skips a level and 'tunnels' into another level. Figure 3.11 shows how tunnels are represented graphically with tunneling into the child diagram (signified by parenthesis on the arrows near the activity box) on the left and tunneling into the parent diagram (signified by parenthesis on the arrows at the ends) on the right. The key class of the CON-

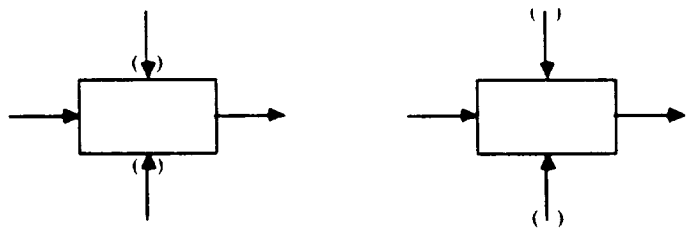


Figure 3.11 ICOM Tunneling Graphical Syntax

CEPT_in_ENV entity class is made up of the concept identifier, the model identifier, and the structure identifier.

3.3.3 Links

A link always starts and ends at a structure; therefore they have been modeled by the metamodel portion shown in Figure 3.12. The LINK entity class contains the starting and ending structures as attributes. In addition, the LINK entity class contains an attribute indicating the concept associated with this link. The key class of the LINK entity class is LID which is a unique symbol to identify a LINK entity.

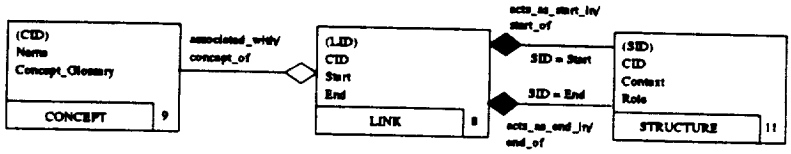


Figure 3.12 Metamodel of Links

3.3.4 Paths

A path relates a producer activity to a consumer activity. Each concept plays a role in the relationship between the two activities. The consumer role can either be a control, an input, or a mechanism. Consequently, the PATH entity class has two weak one-to-many link classes with the ACTIVITY entity class as shown in Figure 3.12. The links represent the producer and consumer activities for this path. The producer and consumer of the path are kept as attributes of the PATH entity class along with an attribute containing the consumer role of the path. Since multiple paths can exist between two activities, the key class of the PATH entity class is made up of a unique symbol (PID).

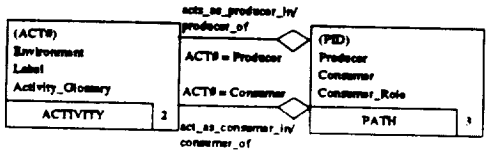


Figure 3.13 Metamodel of Paths

Additionally, Figure 3.14 shows the portion of the metamodel that describes how a collection of links make up a path. Since a link can be a part of many paths (consider a concept flow before it spreads into two separate concepts), the LINK_occ_i_PATH (link occur-

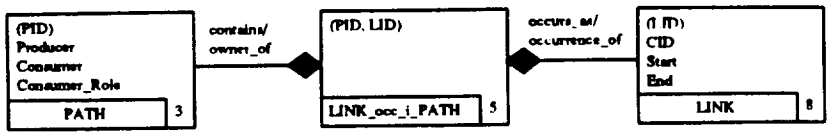


Figure 3.14 Metamodel of Link occurrences in Path

rence in path) entity class is used to represent the situation in which a link is used as part of a path.

3.4 Strengths and Weaknesses of IDEF0

The primary strength of IDEF0 is that the method has proven effective relative to its original structured analysis communication goals for function modeling. Activities can be described by their inputs, outputs, controls, and mechanisms. Additionally, the description of the activities of a system can be easily refined into greater and greater detail until the model is as descriptive as necessary for the decision making task at hand. In fact, one of the noticed problems with models created using IDEF0 is that they often are so concise that unless a reader is an expert in the domain or participated in the model development he (she) will not be able to understand the system that is modeled in the diagrams. The hierarchical nature of IDEF0 facilitates the ability to construct ("AS IS") models which have a top-down representation and interpretation but which are based on a bottom-up analysis process. Beginning with raw data (generally interview results with domain experts) the modeler starts grouping together activities that are closely related or functionally similar. Through this grouping process the hierarchy emerges. If an enterprise functional architecture is being designed (often referred to as "TO-BE" modeling), top-down construction is usually more appropriate. Beginning with the top-most activity, the "TO BE" enterprise can be described via a logical decomposition. The process can be continued recursively to the desired level of detail. When an existing enterprise is being analyzed and modeled, observed activities can be described and then combined into a higher level activity. This process also continues until the highest level activity has been described.

One problem with IDEF0 is the tendency of IDEF0 models to be interpreted as representing a sequence of activities. While IDEF0 is not intended to be used for modeling activity sequences, it is easy to do so. The activities may be placed in a left to right sequence within a decomposition and connected with the flows. It is natural to order the activities left to right because if one activity outputs a concept that is used as input by another activity, drawing the activity boxes and concept connections is clearer. Thus, without intent, activity sequencing can be imbedded in the IDEF0 model. In cases where

activity sequences are not included in the model, readers of the model may be tempted to add such an interpretation. This anomalous situation could be considered a weakness of IDEF0. However, to correct it would result in the corruption of the basic principles on which IDEF0 is based and hence lose the proven benefits of the method. The abstraction away from timing, sequencing, and decision logic allows the conciseness in an IDEF0 model. It also contributes to problems with understanding by readers outside the domain. This particular problem has been addressed with a complementary modeling method called IDEF3.

3.5 Integration With Other Methodologies

The IDEF0 method metamodel is in the process of being integrated with several other method metamodels. The final result of this integration is incomplete, but some discoveries have been made. The metamodels for IDEF0 and Data Flow Diagrams are very similar even though the purpose of the two methods is very different. IDEF0 is intended to model activities while Data Flow Diagrams are intended to model the flow of information. However, it turns out that the information used by each of the methods is quite similar in structure. This similarity will require careful analysis to determine exactly how similar the metamodels are and how they may overlap.

As a counter example, IDEF0 and Structure Charts are two methods that are also very similar in purpose. Their metamodels, however, are not alike at all. Structure Charts model the hierarchy of processes but do not represent their interconnectivity. While IDEF0 also represents hierarchical decomposition, its metamodel contains much more information about the activities by virtue of the inclusion of concept flows. Integration of these two methods will also require careful analysis, but will likely have large parts of the metamodel that do not overlap.

3.6 Conclusions

This chapter has presented a brief description of the IDEF0 modeling method. Also, an IDEF1 model of IDEF0 has been described. By carefully describing the IDEF0 metamodel, it is hoped that the information used by IDEF0 can be integrated with the information used by other modeling methods. Current work by the IDSE research team is progressing towards this goal.

Appendix A. Abbreviations used in the IDEF0 Metamodel

Activity: an entity class.

Activity_Glossary: owned attribute class of Activity, a glossary entry which carries information about the activity and it's function.

Activity_Occ_I_Dec: Activity Occurrence In Decomposition; an entity class.

ACT#: ACTivity number; occurs in the key class of Activity, uniquely identifies an activity.

CID: Concept IDentifier; occurs in the key class of Concept, uniquely identifies a concept.

Concept: an entity class.

Concept_Glossary: owned attribute class of Concept, a glossary entry which carries information about the concept and it's function.

Concept_In_Env: an entity class.

Concept_In_Str: an entity class.

Context: the key class of the parent activity of the decomposition, which serves as the key class of Decomposition.

Consumer: attribute class of Path which identifies the consumer activity.

Consumer_Role: attribute class of Path which identifies the role the consumer activity plays.

Decomposition: an entity class.

End: attribute class of Link which identifies the ending structure.

Environment: attribute class which carries information relative to the environment (the model at the most abstract level) in which the activity belongs.

Explanatory_Text: attribute class of Decomposition containing documentation on this entity class.

LID: Link IDentifier; occurs in the key class of Link, uniquely identifies a link.

Link: an entity class.

Link_Occ_I_Path: Link Occurrence In Path; an entity class.

Model: an entity class.

MID: Model IDentifier; occurs in the key class of Model, partially identifies a model.

Name: attribute class of Concept which captures the name of the concept.

Path: an entity class.

PID: Path IDentifier; occurs in the key class of Path, uniquely identifies a path.

Producer: attribute class of Path which identifies the producer activity.

Role: attribute class of Structure which identifies whether a structure serves as a spread, split, join or merge.

SID: Structure IDentifier; occurs in the key class of Structure, uniquely identifies a structure.

Start: attribute class of Link which identifies the starting structure.

Structure: an entity class.

References

Buffum, H,.E,"Air Force Computer-Aided Manufacturing (AFCAM) Master Plan", Volume III Analytic Tools, AFML-TR-74-104, AFWAL/MLT, WPAFB, OH, 45433.

Ross, Douglas T., "PLEX1: Sameness and the need for rigor, and PLEX2: Sameness and type" Internal Technical Report, Softech Inc., 1975.

Ross, Douglas T., "Software Engineering: Process, Principles, and Goals", *Computer*, May 1975.

Ross, Douglas T., "Structured Analysis(SA): A Language for Communicating Ideas", *IEEE Transactions on Software Engineering*, January 1977.

SofTech,"Integrated Computer-Aided Manufacturing (ICAM) Function Modeling Manual (IDEF0)", Technical Report UM 110231100, June 1981.

4

ENALIM: Conceptual Schema Design

This chapter serves a dual purpose. First, it attempts to describe succinctly the Evolving NAtural Language Information Model (ENALIM) by discussing the history, purpose, syntax, semantics, advantages, and disadvantages of the method. Second, this chapter serves as an integration platform by presenting an Integrated Computer-Aided Manufacturing (ICAM) DEFinition (IDEF) language $IDEF_1$ metamodel of ENALIM and compares common structures of its metamodel with the metamodels of other methods which include $IDEF_0$, $IDEF_1$, $IDEF_{1x}$, ER, and Data Flow Diagrams.

4.1 History and Purpose

An information system consists of three major components:

- functions that retrieve, add, delete, and modify the information base.
- an information base that stores facts about the information system.
- a conceptual schema that contains the rules that describe which information may enter and reside in the information base. It also describes the semantics of the elements in the information base.

A general architecture for an information system [ISO 82] is shown in Figure 4.1. The information system receives a message. The message can either retrieve, add, modify, or delete a piece of information from the information base. The information processor receives the message. The conceptual schema controls the information processor by describing the allowable sentences which may enter the information base. Finally, the information base generates an appropriate message describing the contents of the information base.

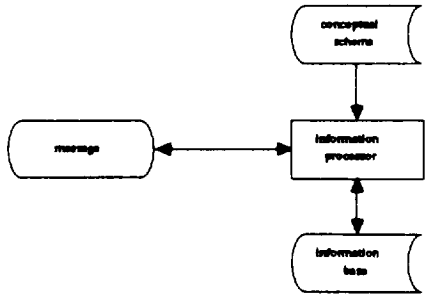


Figure 4.1 A General Information System

In the middle 1970's, Dr. G. M. Nijssen, head of the International Federation of Information Processors (IFIP), developed the concept that information systems are a simplified model of human communication. Consequently, the communication between the environment (the user or application) and the information system can be viewed as a set of natural language sentences for analysis purposes. Using this idea, Nijssen developed the modeling technique of ENALIM for capturing the information needed to design/populate conceptual schemas. ENALIM (today referred to as the Object Role Method) is available today as a part of an information analysis methodology called NIAM (Nijssen Information Analysis Methodology).

4.2 Syntax and Semantics

An ENALIM model is made up of three constructs: 1)object types, 2) fact types, and 3) constraints. An object type is a collection of objects grouped together in order to be compared. Object types can be further classified as NOLOTs (NOn Lexical Object Types) and LOTs (Lexical Object Types). These two classification will be described in more detail in sections 4.2.1 and 4.2.2, respectively. A fact type, which is an association (fact) between two objects, will be described in section 4.2.3. In addition, the constraints (integrity rules) which place restrictions on the population of object types and fact types have been divided into two sections: role constraints and subtype constraints. They will be discussed in section 4.2.4 and section 4.2.5, respectively.

4.2.1 NOLOT (NOn Lexical Object Type)

A NOLOT is an ENALIM object type which denotes a concept or physical object perceived in the universe of discourse but which cannot be directly processed by an information system. The real world objects represented by NOLOTs are presumed to have an

existence independent from a particular naming convention, (i.e. they are not readable or printable). A NOLOT is represented graphically by a circle containing the name (Figure 4.2).

Figure 4.2 Examples of NOLOTs

Two NOLOTs can be related by a subtype ("is a") link. A subtype link is represented graphically by a directed line segment pointing from the subtype to the supertype. The interpretation of the subtype link is that instances of the subtype are instances of the supertype. An instance of the subtype inherits all of the properties of the supertype. The subtype link structure resulting from a model must be acyclic, hence it forms a tree structure. If a tree is made up of *n* NOLOTs, then the tree is called an *n*-NOLOT family. A 3-NOLOT family is shown in Figure 4.3.

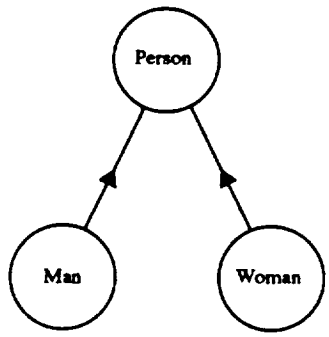


Figure 4.3 An example of a 3-NOLOT family

4.2.2 LOT (Lexical Object Type)

A LOT is an ENALIM object type that represents a real world object which can be passed to and from the information system. This implies that objects represented by LOTs are processable (readable and printable) by the information system. A LOT can refer to, identify, or name a NOLOT. A LOT is represented graphically by a dashed circle containing the name of the LOT as in Figure 4.4.

4.2.3 Fact Types

A fact type is an association (fact) between two object types. Each object type in a fact type association is said to play a role. A fact type

Figure 4.4 Examples of LOTs

is graphically represented by two adjacent rectangles with a line extending from each rectangle to the object associated with the role contained in that rectangle. The only allowable fact types are idea types and bridge types. An idea type is a fact type between two NOLOTs (Figure 4.5). A fact type between a NOLOT and a LOT is called a bridge type, as shown in Figure 4.6.

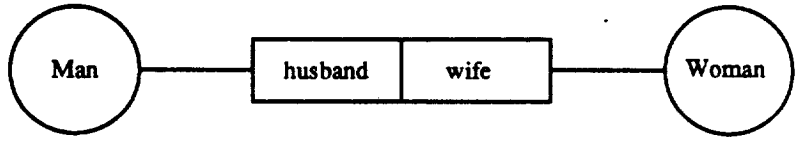


Figure 4.5 An example of an idea type

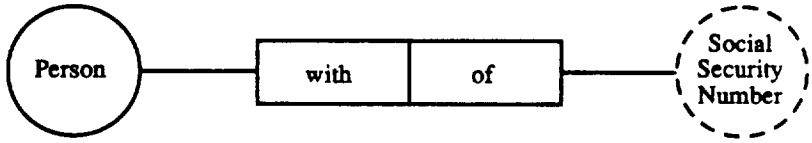


Figure 4.6 An example of a bridge type

4.2.4 Role Constraints

The role constraints place restrictions on the population of object instances for a particular set of roles. The role constraints are the identifier, role uniqueness, total role, role equality, role exclusion, and role subset constraints. These role constraints will be discussed in sections 4.2.4.1 through 4.2.4.6.

4.2.4.1 Identifier Constraint

An identifier constraint (uniqueness constraint or "only one" constraint) declares that a set of object role pairs uniquely identifies an instance of the fact type. An identifier constraint is graphically represented by a dashed line with arrows on both ends ranging over a set of roles in a fact type. In the simple case of binary relationships, four types of identifier constraints are possible: 1) one-to-one, 2) synonym, 3) homonym, and 4) syno-homonym.

The one-to-one identifier constraint declares that either object instance in the constrained fact type can be used to identify the other object instance, and vice versa. In other words, there exists a one to one relationship of an object instance of one role to an object instance of another role, as depicted in Figure 4.7.

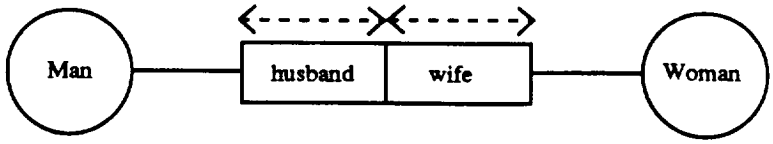


Figure 4.7 A man and a woman participate in only one marriage (monogamy).

The synonym identifier constraint states that an object instance of the first role uniquely identifies an object instance of the second role. Consequently, the synonym identifier constraint represents a one to many relationship from the first object type to the second object type (Figure 4.8).

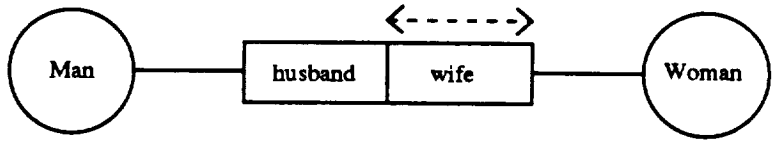


Figure 4.8 A man can have multiple wives, and a woman can have only one husband (polyandry).

The homonym identifier constraint asserts that an object instance of the second role uniquely identifies an object instance of the first role. As illustrated in Figure 4.9, a many to one relationship exists between the object type MAN and the object type WOMAN.

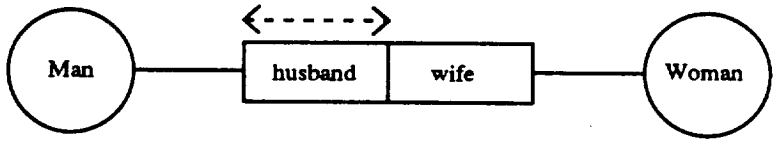


Figure 4.9 A woman can have multiple husbands, and a man can have only one wife (polygyny).

Finally, the syno-homonym identifier constraint states that neither an object instance of the first role nor an object instance of the second role is enough to identify the other object instance. The syno-homonym identifier constraint represents a many to many relationship from the first object type to the second object type as shown in Figure 4.10.

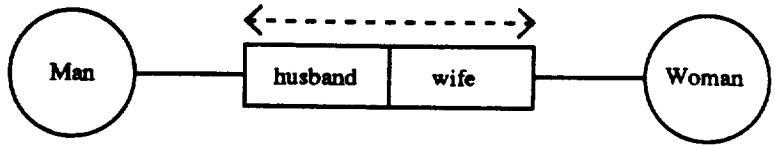


Figure 4.10 A man and a woman may participate in multiple marriages (polygamy).

4.2.4.2 Role Uniqueness Constraint

The role uniqueness constraint specifies that the combination of two or more roles uniquely identifies an object. The role uniqueness constraint is graphically represented by the letter "U" inside of a circle with dashed lines extending from the circle to each role participating in the constraint. As depicted in Figure 4.11, the first name and the last name uniquely identifies an employee.

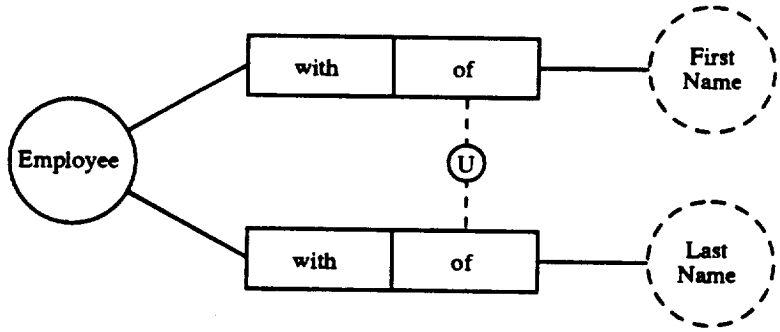


Figure 4.11 A role uniqueness constraint example

4.2.4.3 Total Role Constraint

The total role constraint ("always" constraint) states that there must be an instance of the role for every object type playing that role. This constraint is represented graphically by the universal quantifier symbol appearing on the line between the object type and its role. The total role constraint that a person always has a gender is represented in Figure 4.12.

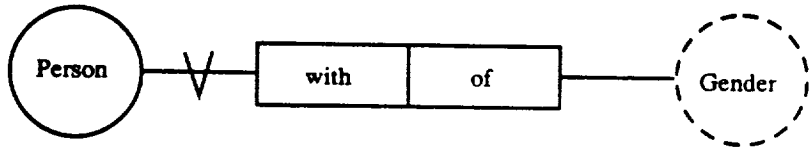


Figure 4.12 A total role constraint example

4.2.4.4 Role Equality Constraint

The role equality constraint states that the set of instances of two roles must be equivalent. The role equality constraint is graphically repre-

sented by the equal sign, "=", inside of a circle in the middle of a dashed line segment connecting two roles. An example of a role equality constraint is represented in Figure 4.13, which states the set of employees working for a department is equivalent to the set of employees earning a salary.

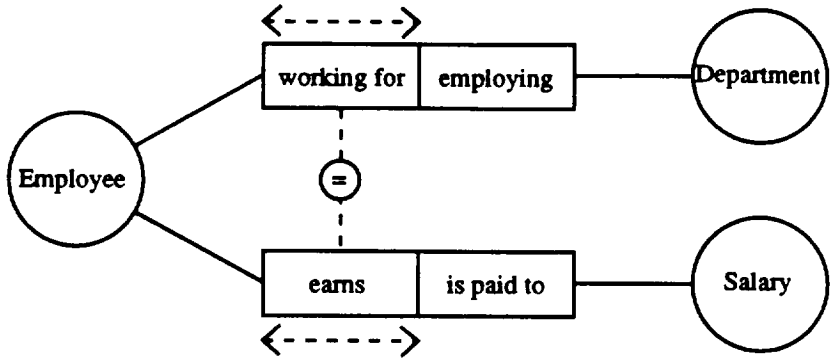


Figure 4.13 A role equality constraint example

4.2.4.5 Role Exclusion Constraint

The role exclusion constraint prescribes that the set of instances of two roles must be mutually exclusive. In other words, an instance of one role cannot appear as an instance of another role. The role exclusion constraint is represented graphically by the letter "X" inside a circle in the middle of a dashed line segment connecting the two roles. As depicted in Figure 4.14, the set of persons earning a salary is disjoint from the set of persons owning a shop.

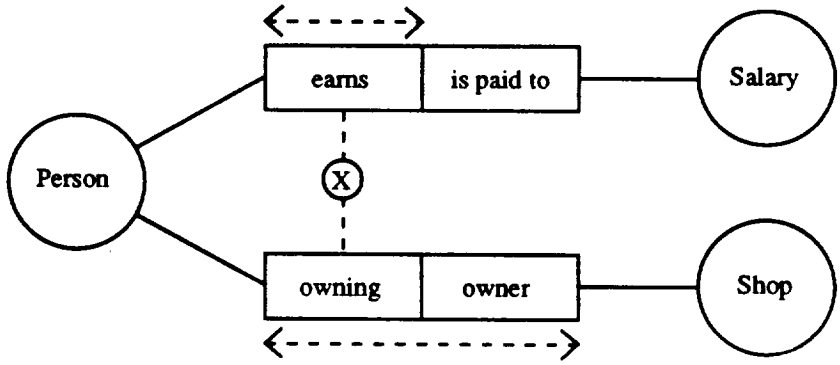


Figure 4.14 A role exclusion constraint example

4.2.4.6 Role Subset Constraint

The role subset constraint states that the set of instances of one role must be a subset of the set of instances of another role. The role subset constraint is represented graphically by a directed dashed line seg-

ment pointing from the subset to the superset. In Figure 4.15, the example states that the set of employees assigned to a project is a subset of the set of employees working for a department.

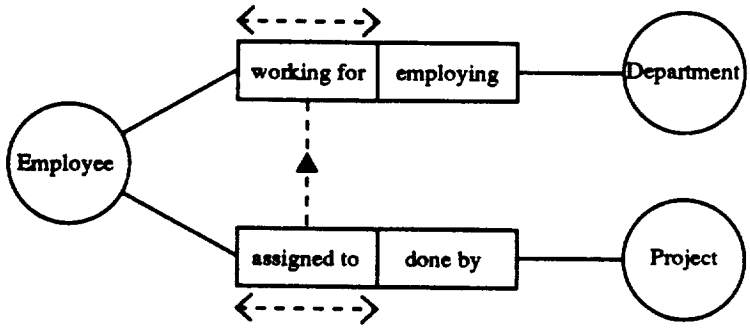


Figure 4.15 A role subset constraint

4.2.5 Subtype Constraints

Subtype constraints restrict the population of the object instances of a supertype into populations of the participating subtypes. The two types of subtype constraints are the subtype exclusion constraint and the subtype total constraint. These constraints will be discussed in section 4.2.5.1 and 4.2.5.2, respectively.

4.2.5.1 Subtype Exclusion Constraint

The subtype exclusion constraint declares that the set of instances of one subtype are mutually exclusive from the set of instances of another subtype. In algebraic terminology, the intersection of the set of instances of one subtype with the set of instances of another subtype is the empty set. The subtype exclusion constraint is represented graphically by the letter "X" inside a circle with dashed line segments connecting the circle to each subtype link participating in this constraint. As illustrated in Figure 4.16, the subtype man of person is mutually exclusive from the subtype woman of person.

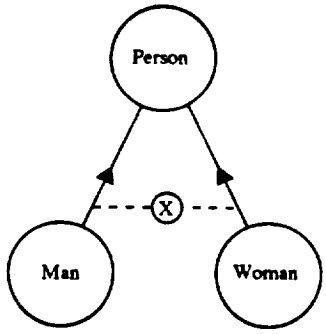


Figure 4.16 Subtype exclusion constraint example

4.2.5.2 Subtype Total Constraint

The subtype total constraint states that the total of all of the instances of one subtype with all of the instances of another subtype make of the set of instances contained in the supertype. In algebraic terminology, the union of the set of instances of one subtype with all of the instances of another subtype make up the set of instances contained in the supertype. The subtype total constraint is graphically represented by the letter "T" inside of a circle with dashed line segments connecting the circle to each subtype link participating in the constraint. An example of a subtype total constraint is the population of men and the population of women which together make up the population of the supertype people, as depicted in Figure 4.17.

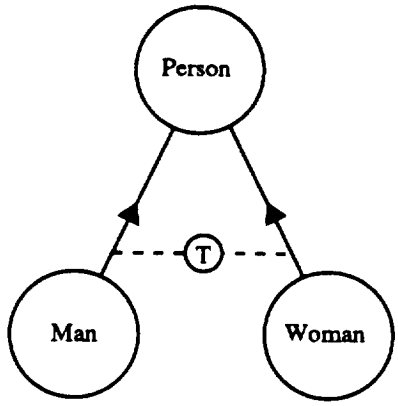


Figure 4.17 A subtype total constraint example

4.3 Metamodel

This section describes an information model of ENALIM (Figure 4.18). $IDEF_1$ is used to model the information contained in an ENALIM model. This information model is referred to as a metamodel. To facilitate the explanation process, the metamodel has been divided into five logical units: 1) NOLOT families, 2) fact types, 3) total role constraints, 4) subtype constraints, and 5) role constraints. The following sections fully describe each of these logical units.

4.3.1 NOLOT Families

The portion of the metamodel that models NOLOT families is shown in Figure 4.19. The entity class OBJECT keeps all of the information about objects. The attribute class OTYPE specifies whether the object is a LOT or a NOLOT. The attribute class ONAME is the name of the object and acts as the key class for this entity class. An additional

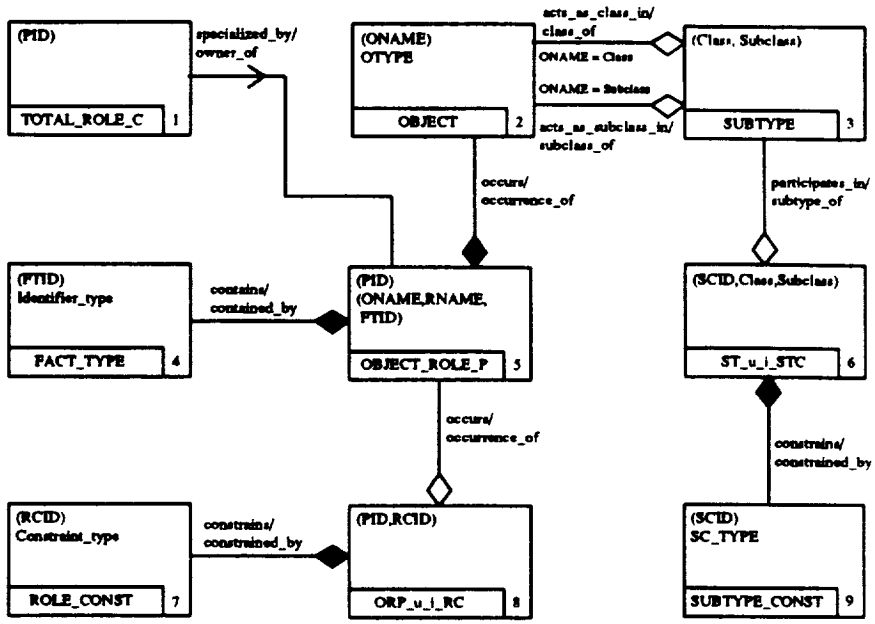


Figure 4.18 IDEF$_1$ Metamodel of ENALIM

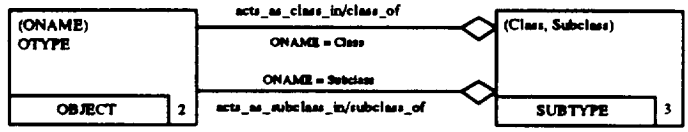


Figure 4.19 Metamodel of NOLOT families

constraint is required to prevent LOT participation in subtype relations. This constraint is represented by the following ISyCL statement:

```
for_all s of entity_class SUBTYPE
    (OTYPE(class_of(s)) = 'NOLOT)
    and
    (OTYPE(subclass_of(s)) = 'NOLOT)
```

The entity class SUBTYPE has as its key class the name of the two NOLOTs contained in this subtype link. This implies that only one subtype link can exist between two individual NOLOTs. An additional link constraint is needed to state that a NOLOT cannot be a subtype of itself and that no matter what NOLOT you visit in a NOLOT family, a path will not exist along the subtype links that will return to the starting NOLOT. In other words, a NOLOT family is a directed acyclic graph. These constraints are represented by the following ISyCL statements:

```
function superclass?(obj1, obj2):boolean
"Is OBJ2 a superclass of OBJ1?"
    [(obj1 <> obj2)
    and
    (for_some s of entity_class SUBTYPE
        (class_of(s) = obj1)
        and
        (subclass_of(s) = obj2))
    and
    (for_some s of entity_class SUBTYPE where (class_of(s) = obj1)
        superclass?(subclass_of(s), obj2))]

for_all s of entity_class SUBTYPE
    "No non-acyclic graphs"
    not (superclass?(subclass_of (s), class_of(s)))
```

4.3.2 Fact Types

As shown in Figure 4.20, every object in a model belongs to at least one object role pair (OBJECT_ROLE_P). The entity class OBJECT_ROLE_P contains the object name, ONAME, and the role name, RNAME, belonging to this OBJECT_ROLE_P. An entity of the entity class OBJECT_ROLE_P is identified by the key class PID, which is a unique symbol. A fact type is made up of two object role pairs. Each fact type has an identifier-type attribute class whose attribute value may be either one-to-one, synonym, homonym, or syno-homonym. A link constraint exists that states that the object type of the two object role pairs participating in a fact type cannot both be LOTs. The only allowable combinations are between a NOLOT and a LOT, which is called a bridge type, or between two NOLOTs, which is called an idea type. This constraint is represented by the following ISyCL definition:

```
for_all f of entity_class FACT_TYPE
    not(for_all p in contains(f, OBJECT_ROLE_P)
        (OTYPE(ONAME(p)) = 'LOT))
```

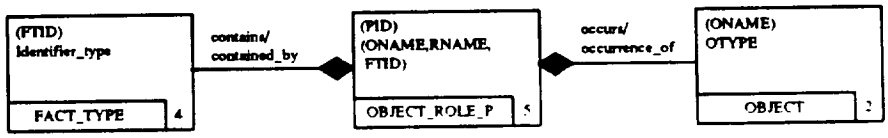


Figure 4.20 Metamodel of fact types

4.3.3 Total Role Constraint

The total role constraint is modeled by the one-to-zero-or-one link class from entity class OBJECT_ROLE_P to the entity class

TOTAL_ROLE_C (Figure 4.21). The total role constraint is modeled as a separate entity class to avoid violating the no null rule of IDEF$_1$, because not every OBJECT_ROLE_P has a total role constraint. The key class of the total role constraint is the key class of the OBJECT_ROLE_P that it is associated with it.

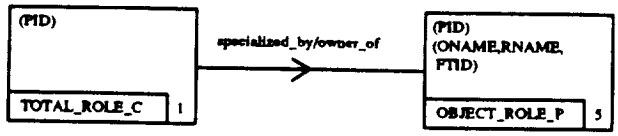


Figure 4.21 Metamodel of total role constraint

4.3.4 Subtype Constraints

The subtype constraints of subtype exclusion and subtype total have been modeled in Figure 4.22. Since a subtype link can appear in multiple subtype constraints and a subtype constraint is made up of multiple subtype links, the entity class ST_u_i_STC (subtype use in subtype constraint) was added to the model.

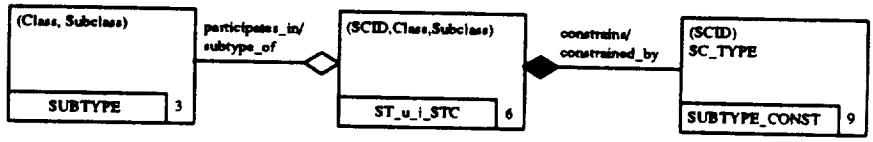


Figure 4.22 Metamodel of subtype constraints

4.3.5 Role Constraints

The role constraints are modeled in Figure 4.23. The role constraints include the joint uniqueness, role equality, role exclusion, and role subset constraints. The entity class ORP_u_i_RC (object role pair used in role constraint) shows the pairwise relationship between a role constraint and each object role pair participating in this role constraint. This entity class was added to the model since an object role pair can participate in many role constraints and a role constraint is made up of many object role pairs.

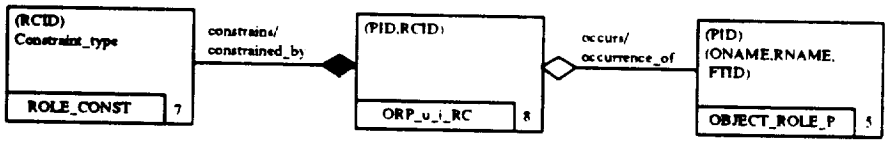


Figure 4.23 Metamodel of role constraints

4.4 Strengths and Weaknesses

Possibly the greatest strength of ENALIM is the fact that it embodies a representation of both the real world objects and their relations along with the data objects and relations into a single integrated syntax. If one takes the meaning of "semantic data model" to be the documentation of the link between the data in an information system and the "things"/"situation" represented by that data in the real world, then ENALIM is the only method we have studied that actually captures both aspects unambiguously.

$IDEF_1$ clearly distances itself from the representation of objects in the real world (i.e. entity classes like "employee" do not represent real world people but other collections of information presumably about the real world object named by the entity class). Both $IDEF_{1X}$ and ER conflate the two, thus making it impossible to tell if an "entity" (in $IDEF_{1X}$) or an "entity set" (in ER) is intended to represent the object itself or the information about that object. ENALIM, with its clear distinction between LOTs and NOLOTs was the first (and to date only) method to grapple with trying to simultaneously represent and keep both concepts distinct.

ENALIM's strength resides in the fact that it is based on the deep structure of sentences. The rich set of constraints gives ENALIM the ability to capture all nuances of a sentence. In addition, all the sentences and constraints of ENALIM have a graphical notation with text needed only in rare occasions.

Being first is not always an enviable position. ENALIM does suffer from a bit of impoverishment in its ability to describe situations in the real world component. Deficiencies in the information modeling component have been addressed with subsequent $IDEF_1$ like additions under the NIAM method set. However, major deficiencies in the real world component relative to representing abstractions, temporal relation, definite descriptions, and others have received little formal treatment.

The lack of a focusing mechanism is ENALIM's primary deficiency. Instead of being able to describe details of a portion of the model and then hide these descriptions at a higher level of abstraction, the model is made up of only one level of detail. Therefore, models tend to explode in size and becomes unmanageable even with currently available automated tools. The above developed information metamodel of ENALIM will be used to provide some insight into ways of alleviating this problem.

4.5 Tips and Traps

The main trap analysts tend to fall into is that they do not constrain the enterprise they are modeling. Therefore, the models tend to become extremely large. Consequently, an ENALIM model must be properly focused on the information system to be modeled. This will decrease the model size and corresponding complexity.

4.6 Integration With Other Methodologies

The IDSE Research Team is currently looking for commonality among the previously mentioned methods based on each methods' metamodel. Once the equivalent model constructs can be determined, a neutral information representation schema will be developed. At this writing, we are still in the process of determining the common constructs across the different methodologies.

4.7 Conclusions

A concise description of the ENALIM methodology has been included to aid in the description of the $IDEF_1$ metamodel of ENALIM. This metamodel serves as the basis from which integration decisions concerning ENALIM will be derived. Additional benefits of the metamodel include: 1) providing a less ambiguous understanding of the methodology among the team members, 2) providing a common reference point for the team from which decisions can accurately be made concerning integration, and 3) providing an initial platform for the development of integration techniques.

Appendix A. Abbreviations used in the ENALIM Metamodel

Class: inherited attribute which partially identifies a subtype.

Constraint-Type: attribute which describes the type of constraint. Codomain: joint uniqueness, role equality, role exclusion and role subtype constraint.

RCID: Role Constraint IDentifier; uniquely identifies Role-Constraint.

Fact-Type: an entity class which describes the association between two objects.

Identifier-Type: attribute which specifies the categories of identifier constraints. Codomain: one-to-one, synonym, homonym and syno-homonym.

Object: an entity class which keeps information about object types.

Object-Name: occurs in the key class of Object, uniquely identifies the object.

Object-Role-Pair: an entity class describing the role an object plays in a relation.

Object-Type: attribute which specifies the type of object - LOT or NOLOT.

ORP-u-i-RC: Object Role Pair used in Role Constraint; an entity class.

PID: Pair IDentifier; uniquely identifies Object-Role-Pair.

Role-Constraint: an entity class which describes constraints on the object instances for a set of roles.

Role-Name: attribute which identifies the role an object plays in Fact-Type.

SCID: Subtype Constraint IDentifier; occurs in the key class of Subtype-Constraint, uniquely identifies Subtype used in Subtype Constraint.

ST-u-i-STC: SubType use in SubType Constraint; an entity class.

Subclass: inherited attribute which partially identifies a subtype.

Subtype: an entity class which describes the subtype link.

Subtype-Constraint: an entity class which identifies the type of constraint placed on the subtype.

Total-Role-Constraint: an entity class.

References

Nijssen, G. M., "The Next Five Years in Data Base Technology", Paper presented at: Infotech State of the Art Conference, Regent Centre Hotel, London, 12-14 December 1977.

Nijssen, G. M., "On Conceptual Schemata, Databases, and Information Systems", Preliminary Version, Paper presented at: Data Bases - Improving Usability and Responsiveness, August 2-3, 1978, Haifa, Israel.

ISO, Concepts and Terminology for the Conceptual Schema and the Information Base edited by J. J. van Griethuysen, March 15, 1982.

Nijssen, G. M., Informatie Analyse en Data Bases 82, Universiteit van Queensland, Brisbane, Australie, December 1982.

Nijssen, G. M., De Productieve Combinatie ISAC + NIAM = 3, Universiteitvan Queensland, Brisbane, Austriale, December 1982.

Thompson, Paul, "Natural Language Analysis, Information Modeling, and Database Engineering", Control Data Corporation, Minneapolis, Minnesota, February 14, 1985.

Van Assche, F., "Some Natural Extensions to NIAM", International Center for Information Analysis Services, Control Data Belgium, Inc., September 1985.

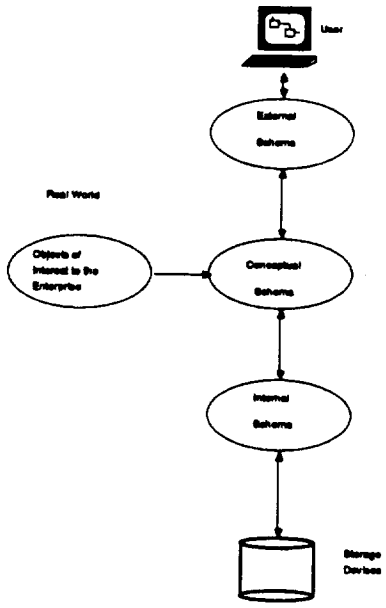
5

IDEF$_{1X}$: Data Modeling

Data modeling is one facet of the overall Information Systems Architecture (ISA) development scheme. Several methodologies for data modeling, including the Integrated Computer-Aided Manufacturing (ICAM) DEFinition (IDEF) language IDEF$_{1X}$, Chen's Entity-Relation (ER) [Chapter 6], and Nijssen's Evolving NAtural Language Information Model (ENALIM) [Chapter 4], have emerged over the past fifteen years. Historically, data modeling was introduced for database design. Consequently, the developers of these methodologies have been influenced by the needs of a database designer. The metamodel of IDEF$_{1X}$ presented in this chapter was developed as part of an effort to integrate a complete set of ISA modeling methods. The metamodel can also be used to aid in understanding the basic concepts and principles of the methodology and to contrast IDEF$_{1X}$ with the other data modeling methodologies.

5.1 History and Purpose

A methodology is a language system. Like any other type of system, there are many different methodologies for various purposes. There are currently three primary IDEF methodologies: IDEF$_0$, IDEF$_1$, and IDEF$_{1X}$. There is also IDEF$_2$ which was developed to support simulation modeling. It has largely been replaced by commercially available simulation modeling systems. IDEF$_0$ is used to model activities and the relations between activities. IDEF$_1$ models the logical structures of the information in a system. Finally, IDEF$_{1X}$ was introduced to model the data kept about entities within a system for the purpose of conceptual schema design for three schema database systems as defined by the ANSI SPARC report on database management systems [DACOM 85, ANSI 75]. Note that this is not the same as conceptual schema design for the conceptual information processor integration concept as defined in the ISO report [ISO 87].



ISO Conceptual-Schema Architecture

Because of the name, IDEF1x is often thought of as an extension to IDEF1. In actuality, the two are complimentary. IDEF1x picks up at the data design point after the information requirements (expressed in IDEF1) are complete. The developers of IDEF1x did not simply extend IDEF1, but instead started from different foundations. For example, as stated in [DACOM 85] IDEF1x entities correspond to "things about which data is kept, e.g. people, places, ideas, events, etc.", in contrast to the IDEF1 entity which corresponds to "logical information managed in the organization." We have used IDEF1 as our metamodeling language for this analysis effort since we must do an information level integration of the methods prior to doing a logical database design. As the IDEF1 model of IDEF1x is developed later in the paper, the differences between the two methodologies will be demonstrated.

The primary reference for IDEF1x is the Integrated Information Support System (IISS) report prepared for General Electric by the D. Appleton Company [DACOM 85]. That report provides a brief history, a thorough review of the syntax and practice, and then a detailed description of how to build an IDEF1x model. A formal theoretical foundation (syntax and semantics) for the method was published in an Integrated Information Systems Evolution Environment (IISEE) Report [Mayer 88].

The purpose of this report is not to duplicate what was done in the previous reports, but instead to describe the information managed within an IDEF1x model by building an information model of it. The purpose of this metamodel is to provide the basis for determining how to integrate IDEF1x with other modeling methodologies. Until multiple methodologies can be integrated, there cannot be a coherent framework for system development or a truly useful integrated development support environment.

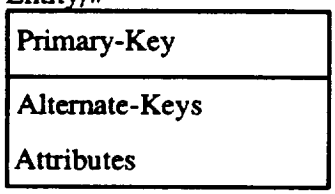
5.2 Syntax and Semantics

There are two stages of learning to model. The first is learning the syntax and semantics of the modeling methodology. This is usually done by having an expert teach a short course. On the other hand, since the IDEF methodologies are syntactically easy to learn, it is possible to learn their syntax independently. The following section should go a long ways toward that goal for those unfamiliar with the IDEF1x method.

Once the syntax and semantics are understood, the hard part begins (which is generally the reason for engaging an expert). Modeling can actually be considered an art. It generally requires a large amount of considered judgement. It is easy to create a meaningless (or blatantly wrong) model. Each step of the modeling process, particularly the model validation, needs to be followed carefully, so that the completed model is consistent. It is beyond the scope of this work to teach proper modeling techniques, but where possible tips will be given. It is also a goal of the IDSE Project to develop tools that will aid in checking the semantics of a model.

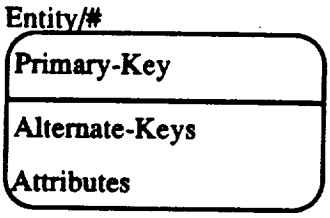
5.2.1 Entities

An entity represents a set of data instances. For example, the entity "Person" represents the data kept about people in an enterprise. The instances could be data kept about Jim, Mary, or Bob. Similar data are kept about each of the instances. It is important to keep in mind that an "entity" represents sets of data, not the physical objects that the data describes.



Identifier-Independent Entity

There are two primary types of entities, identifier-independent and identifier-dependent. Identifier-independent entities can exist without any other entities, while identifier-dependent entities are meaningless without other entities. In a model of graduate students, the student's committee is an example of an identifier-dependent entity. The committee is dependent on the student and his or her advisors



Identifier-Dependent Entity

for its existence. Dependence and independence are specific to a model.

Identifier-independent entities are represented by rectangles with square corners. The unique entity name is placed just above the box along with a unique entity number. The box is divided by a solid line. The primary set of attributes which uniquely identify the entity are placed above the line.

Identifier-dependent entities look similar to identifier-independent entities, except that the corners of the rectangle are rounded. Identifier-dependent entities inherit at least one of their primary key attributes from a parent entity.

5.2.2 Connection Relationships

Connection relationships show how entities (sets of data instances) relate to one another. The relationships are always between exactly two entities. The connection relationship starts at the independent, or parent, entity and ends at the dependent, or child, entity. The connection relationship is labeled with a verb phrase which describes the relationship. A filled circle is drawn at the dependent end.

The connection relationship in Figure 5.1 is called an identifying relationship. Identifying relationships are signified by a solid line. Non-identifying relationships are drawn as a dashed line. The child entity in an identifying relationship must be identifier-dependent.

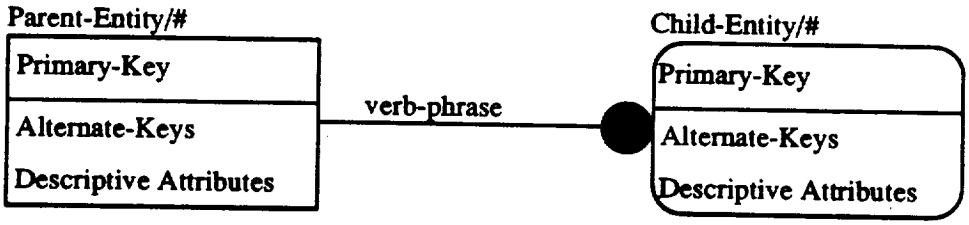
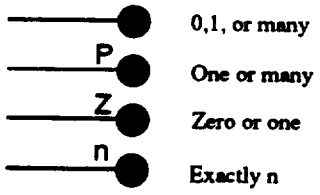


Figure 5.1 Identifying Connection Relationship



Cardinalities

Each connection relationship has a cardinality. The cardinality specifies the number of instances of the dependent entity that are related to an instance of the independent entity. For example, an instance of the data about a house is related to many instances of the data about a room.

There are four different cardinality types. Relations are always drawn starting at the independent entity. Thus, a zero-or-one relation means that there is zero or one dependent entity for every one independent entity.

5.2.3 Categorization Relationships

Up until now IDEF1x has been similar to IDEF1 syntactically. Categorization relations are specific to IDEF1x. They cause models developed in the two methodologies to look quite different.

Categorization relationships allow the modeler to define categories of objects. For instance there could be an entity named "Car" which is the generic entity in a category showing different types of cars. Each of the category entities must have the same primary key as "Car". Also, there must be a way of distinguishing between the category entities. The category entities are distinguished by a discriminator attribute which must have a different value for each category entity. The category relationship syntax is shown in Figure 5.2.

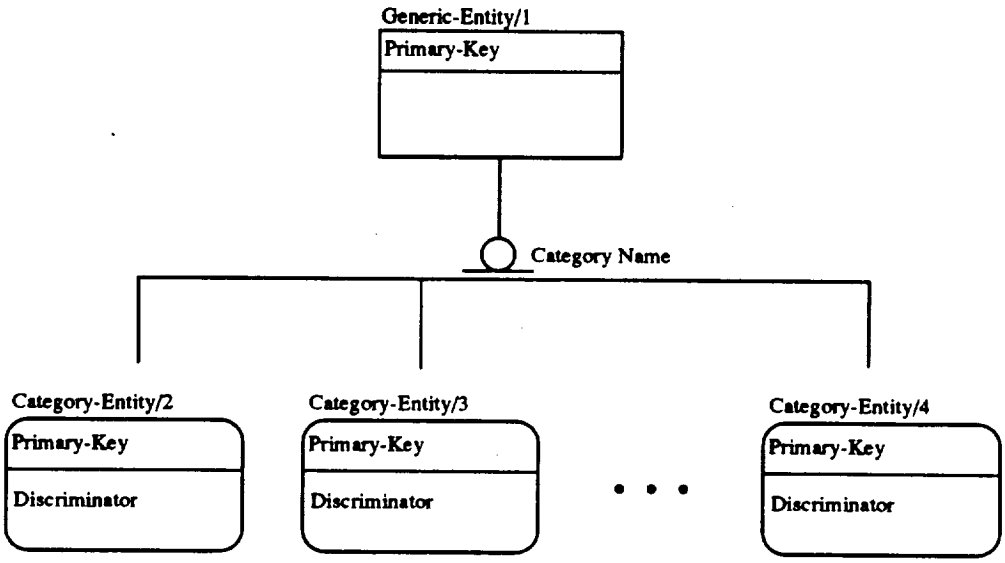


Figure 5.2 Category Relationship Syntax

It is important to make sure that there is a need for a category, and that meaningless entities are not being created by mistake. Some models have category entities which do not contain the discriminator attribute. Though this may be reasonable in some cases, it can lead down the path toward unnecessary entities.

An entity can act as a generic entity in many category relationships, but an entity can only act as a category entity in one relationship. Also, a category entity can have only one generic entity. In other words, hierarchies must be structured so that it is not possible for an entity to be a member of two categories.

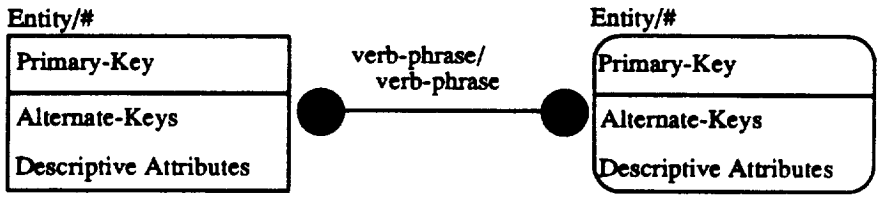
Since a category entity is only a category member, it cannot participate as a child in an identifying connection relationship. Only the generic entity can participate in such a relation.

5.2.4 Non-Specific Relations

In the process of developing a model it is sometimes necessary to admit not understanding the information to be modeled. By the time the model is complete, the misunderstandings can be cleared up, and a proper model presented. Non-specific relations are a vehicle for this type of development.

Non-specific relations are many-to-many relations. Each end of the relation has a cardinality. Also, two labels are placed on the link corresponding to the two directions of the relation. For instance, if one needed to model organizations and their members, it could be said that an organization consists of many people and that a person can participate in many organizations.

The same rules apply as for normal connection relations plus there is the condition that all non-specific relations must be replaced before release of the model.



Non-Specific Relation

In non-specific relations, any cardinality may be used at either end of the link, and the verb-phrase on top refers to the relationship from left to right and the bottom verb-phrase refers to the relationship from right to left.

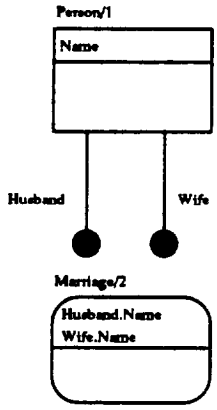
5.2.5 Attributes

Attributes contain information which is used to describe an entity. Attribute names are unique throughout an entire IDEF1x model, and the meaning of the names must be consistent. For example, the attribute "color" could have several possible uses. "Color" could mean hair color, skin color, or a color in a rainbow. Each of these uses has a range of meaningful values, and thus should be named more clearly (e.g. "hair-color").

Every attribute is owned by exactly one entity. The attribute "SSN" (Social Security Number) could be used in many places in a model but would probably be owned by the entity "Person". Attributes are inherited across relations, thus they can be used in many entities other than their owner.

Every attribute must have a value (No-Null Rule), and no attribute may have multiple values (No-Repeat Rule). These rules enforce the creation of proper models. If there is a situation where it seems that one of these rules needs to be broken, then the model is likely wrong.

Attributes are displayed inside entity boxes as shown in previous figures.



Role Names

5.2.6 Role Names

There are cases where the same attribute will be inherited from different relations. In such cases it is aids clarity to append a role name to the front of the attribute name. The role name is appended to the front of the attribute name with a period between.

For example, two people participate in a marriage, and if Name were inherited from Person to Marriage it would be convenient to append Husband on to the name inherited from the man and Wife on to the name inherited from the woman.

5.2.7 Keys

A key is a grouping of attributes which uniquely identify an instance of an entity. There are primary and alternate keys. Every entity has exactly one primary key and it is displayed as the set of attributes above the horizontal line in the entity box. Entities can also have alternate keys which also uniquely identify the entity, but are not used for describing relationships with other entities.

In a connection relationship, the primary key of the parent migrates to the child. If the relationship is a category relation, then the primary key of the child is the same as the generic. If the relationship is an identifying relationship then the primary key of the child must contain attributes inherited from the parent.

Attributes which participate in alternate keys are designated by "(AK#),'' where the # is the number of the alternate key. To find the attributes in an alternate key, each attribute is checked to see if it participates in that alternate key. Attributes may participate in many keys, so there could be more than one "AK#" in the list beside an attribute.

Besides the fact that a key must uniquely identify an entity, all attributes in the key must contribute to the unique identification (Smallest-Key Rule). Thus, when deciding whether or not an inherited attribute should be made part of a key, it must be decided whether that attribute is necessary for unique identification. It is not sufficient to say that it contributed to the unique identification of the parent.

There are also two dependency rules. First, there is the Full-Functional-Dependency Rule. This states that if the primary key is composed of multiple attributes, then all non-key attributes must be functionally dependent on the entire primary key. Second, is the No-Transitive-Dependency Rule. It states that every non-key attribute must only be functionally dependent on key attributes.

5.2.8 Foreign Keys

Foreign keys are not really keys at all, but attributes inherited from the primary keys of other entities. Foreign keys are labeled with an "(FK)" to show that they are not owned by that entity. Foreign keys are significant in that they show the relationships between entities. Since entities are described by their attributes, if an entity is composed of attributes inherited from other entities, then that entity is similar to those entities.

5.3 Metamodel

In order to understand the information contained in an $IDEF_{1x}$ model, a metamodel of $IDEF_{1x}$ in $IDEF_1$ has been constructed. Developing a metamodel is a tricky process. It is important to differentiate between semantic and syntactic information. The syntax of $IDEF_{1x}$ has been presented in the previous section. While the semantic rules of $IDEF_{1x}$ were also presented earlier, this section will go deeper.

The metamodel contains eleven entity classes (Figure 5.3). The plan of attack for describing the metamodel is to divide it into submodels in order to reduce the complexity of the model. After looking at each submodel, model-wide issues will be addressed.

5.3.1 Entity Submodel

The most important entity class is *Entity*. Since entities are the actual data objects, it is intuitive that the other entity classes will be to some degree dependent on *Entity*. It was stated earlier that there are two types of entities, identifier-independent and identifier-dependent. Being identifier-dependent means that the entity participates as a child in at least one identifying relationship. By tracing down the

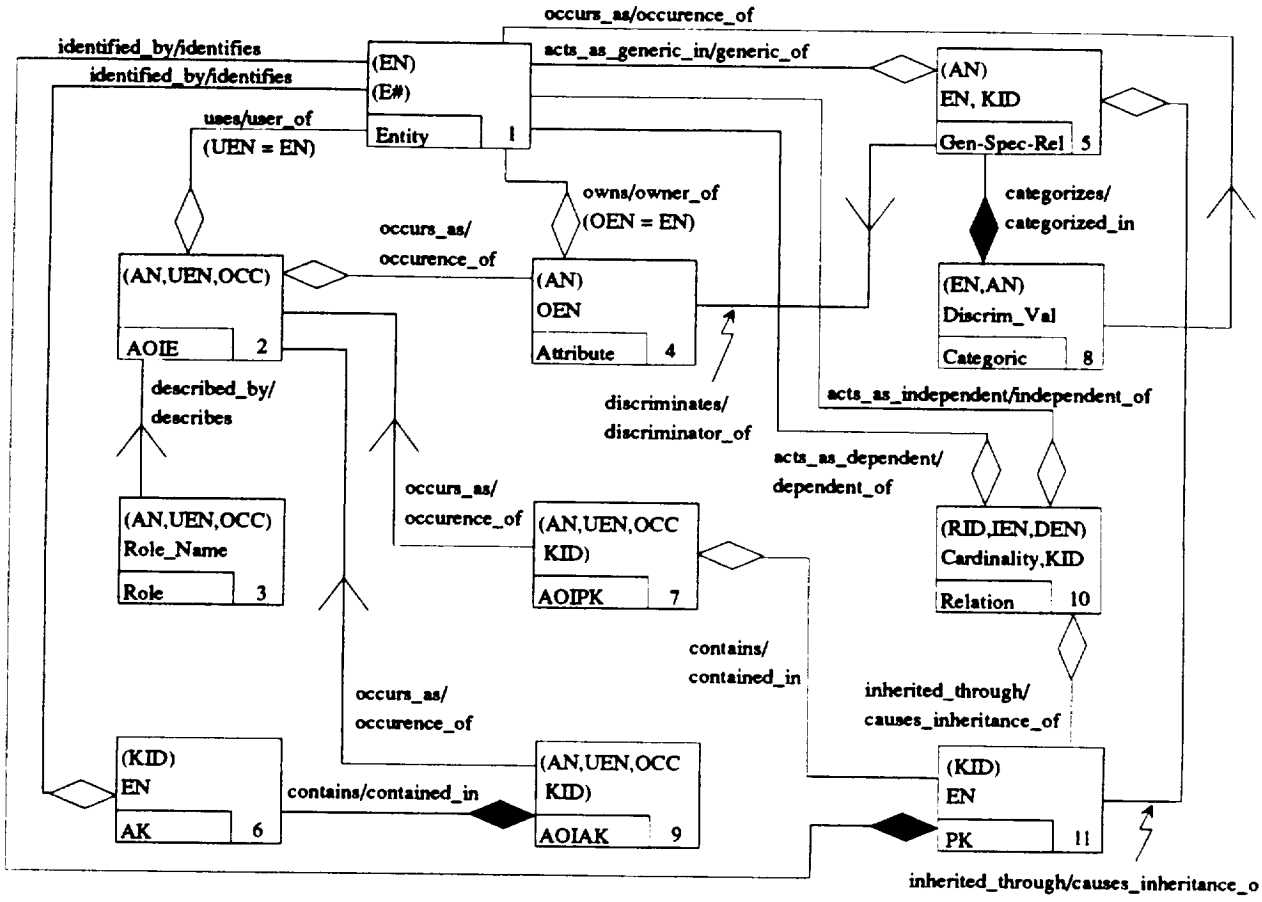


Figure 5.3 Metamodel of $IDEF_{1x}$ in $IDEF_1$

acts_as_dependent link to *Relation*, it can be determined whether the relationships in which the entity participates as a child are identifying relationships by comparing the attributes in the key which is inherited through the relationship (identified by KID) with the entity's primary key.

Let us take a moment and discuss attributes in $IDEF_1$. In the metamodel of $IDEF_{1x}$, *Entity* could have an attribute describing whether it is identifier-dependent or identifier-independent. The question to ask is whether or not that attribute would add meaningful information to the model. In this case it would not because the information is already represented by its link to *Relation*. If this metamodel were used to implement a tool for $IDEF_{1x}$, it is quite likely that a designer would add such a field to his data structure and database schema to reduce the time it would take to determine an

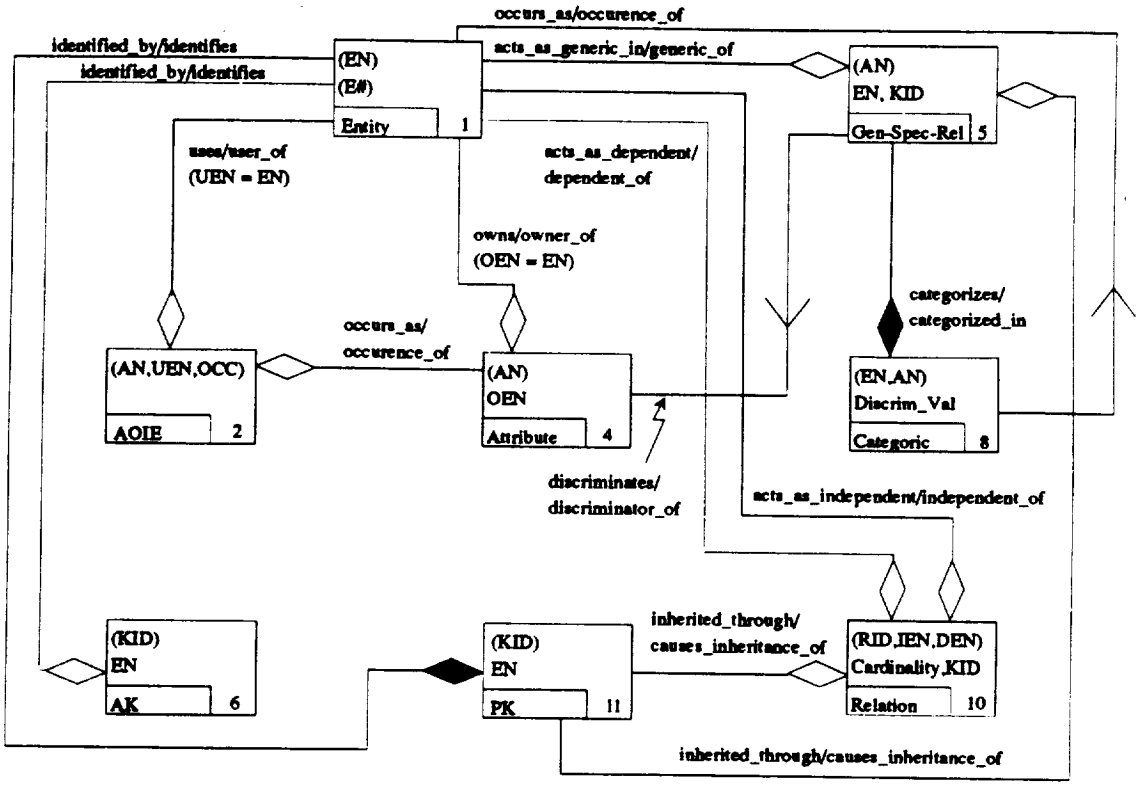


Figure 5.4 Entity Submodel

IDEF1x entity's dependency. The current metamodel is not describing an implementation, but instead the information present in an IDEF1x model.

The key class of *Entity* is "simple." Simple means that there is only one attribute class in the key class. Every entity in IDEF1x has a unique name, thus the entity's name is enough to uniquely identify the entity.

Enough with the aside, let us get back to the main course. The representation of the relationships between entities and attributes in IDEF1x is more complex than might be expected. First, every attribute is owned by exactly one entity. This relationship is described by the link between *Entity* and *Attribute*. An entity can own zero, one, or many attributes. Note the label above the link, *OEN = EN*. This means that the entity name (*EN*) is inherited by *Attribute*, but the name is changed to *OEN* (owned entity name) to signify that entity is the owner of that attribute.

Entities also contain one or more attributes. One might at first draw a strong many-to-one link from *Entity* to *Attribute*. However, attributes can participate in many entities. This would mean a many-to-many link between *Entity* and *Attribute*, which is not allowed. The

solution to this dilemma is a new entity class for attributes occurring in entities (*AOIE*). The appropriateness of this solution will be seen later when roles are discussed.

Along with attributes, entities have groups of attributes called keys which uniquely identify instances of the entity. An entity must always have a primary key, thus the strong many-to-one link from *Entity* to *PK* (Primary Key). There may also be alternate keys (*AK*) which uniquely identify the entity. Only the primary key is inherited across relations though. The entity name (*EN*) is inherited by *PK* as a non-key class attribute. The reasoning for this is that every key is unique, but there may be more than one key per entity or a key may participate in more than one entity. Thus the entity name is not sufficient to uniquely identify a key. A Key-ID (*KID*) is generated to uniquely identify the key.

Note that an entity can only have one primary key, but we show a strong-many-to-one relationship between *Entity* and *PK*. Surely, some gyrations could be done to try and express the constraint that an entity must have exactly one primary key, but would it serve instead to just make the model unreadable? There is a cleaner approach. As part of the IISEE Project, a constraint language based on first-order predicate calculus has been developed to handle this type of situation. Actually, the constraint language is powerful enough to describe all of IDEF1 as well.

The constraint necessary to constrain an entity to one primary key would be written:

```
for_all e of entity_class:Entity
    length (identified_by(e,PK)) = 1;
```

which checks to see whether entities have exactly one primary key. *Identified_by* returns the set of primary keys which identify the entity. This should be a singleton set.

Entities can participate in category relations as generic entities or category entities. Categoric entities are identified by the *Categoric* entity class. There are one or many categoric entities in each generalization/specialization relationship (*Gen-Spec-Rel*). Generalization/specialization relationships can be identified by the attribute which acts as discriminator in the relationship. The generic entity is identified by the *EN* attribute class inherited through the *acts_as_generic_in* link class. Again, a constraint is needed which specifies that an entity cannot act as both generic and categoric entity in the same category relationship. The constraint would be written:

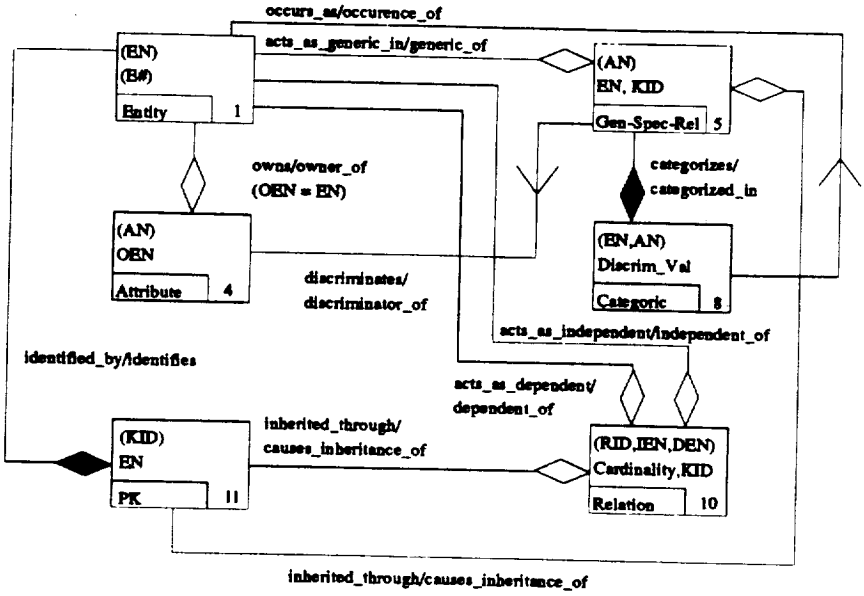


Figure 5.5 Relation Submodel

```
for_all e of entity_class:Entity
    for_all c in occurs_as(e, Categoric)
        (EN(categorized_in(c)) <> EN(e))
```

which checks (for all entities) the set of relations in which the entity acts as generic in to make sure there are no relations in which the entity acts as both generic and categoric. The constraint specifically states, checking all entites (first line), that for all occurences of an entity as a categoric (second line), the generic entity of the generalization/specialization relationship should not be the entity in question (third line).

5.3.2 Relation Submodel

Relations are represented by links between entities which are labeled with a verb phrase. There can be many relationships between two entities, so in addition to the Dependent Entity Name (*DEN*) and Independent Entity Name (*IEN*), a Relation ID (*RID*), which is the verb phrase used to label the link, is also used to uniquely identify the relation.

All relations cause the primary key of the independent entity to be inherited by the dependent entity. In the case of identifying relationships, some or all of the inherited attributes must be used in the primary key of the dependent entity. In a non-identifying relationship, attributes from the primary key of the independent entity are inherited by the dependent entity, but none of them may be used in the primary key of the dependent entity.

5.3.3 Key Submodel

Keys are collections of attributes that are responsible for uniquely identifying entities and showing the inheritance of attributes between entities. Many times there are many sets of attributes which could uniquely identify an entity. The choice of which set becomes the primary key is made by deciding what information should be passed to other entities.

Every entity has one or more keys. Keys are uniquely identified by an autogenerated Key ID (*KID*). If there is only one key, then that key is the primary key. The entity name (*EN*) of the entity which contains the key is also kept.

Keys are made up of one or more attributes, and an attribute can participate in many keys. Consequently, the Attribute Occurrence In Key (*AOIxK*) entity is used to describe this many-to-many relationship. The keys of Attribute Occurrence In Entity (*AOIE*) and *PK* or *AK* are combined to form the key class of the *AOIxK* entities.

5.3.4 Attributes and Roles

Attributes have already been discussed in some detail in the previous submodels, but the subject of attribute roles needs to be addressed.

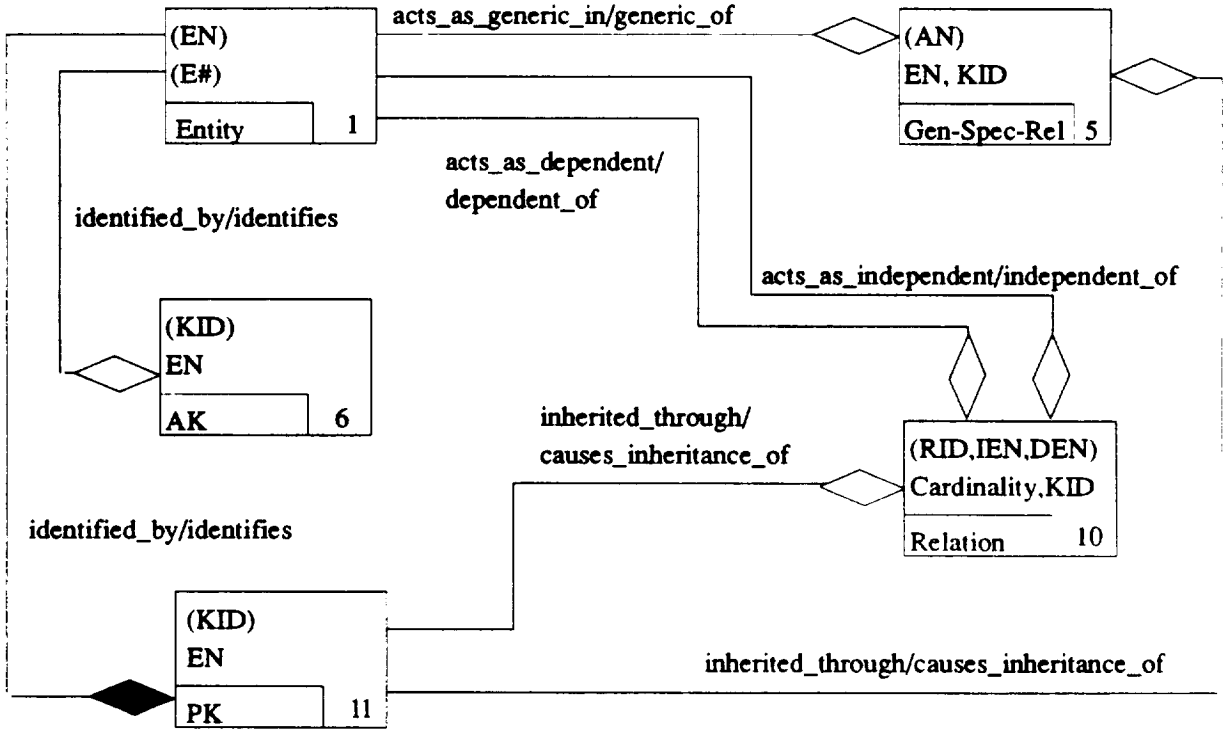


Figure 5.6 Key Submodel

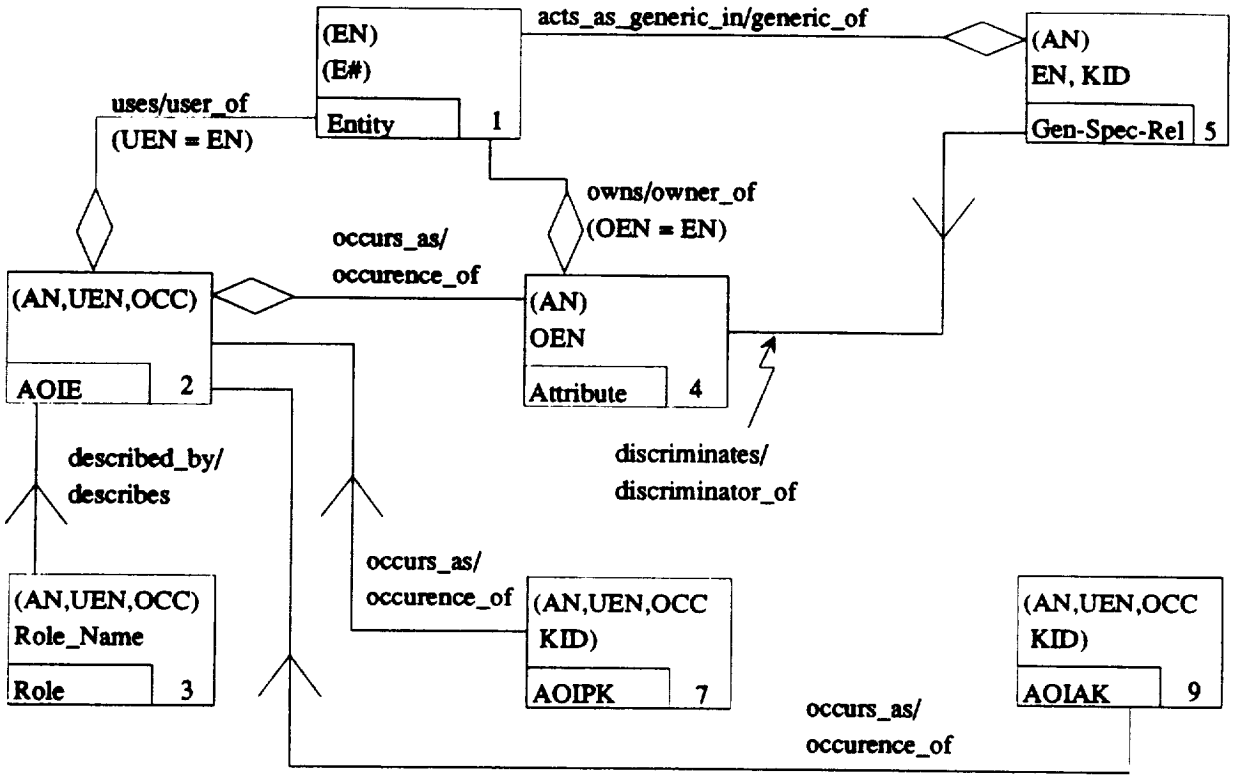


Figure 5.7 Attribute Submodel

As was discussed earlier, it is often convenient to append a role name on to the front of an attribute to show which relationship caused the inheritance of that attribute.

Since there is no particular information that needs to be kept about a role name, it might seem that it could be an attribute of Attribute Occurrence In Entity (*AOIE*). There is a fatal flaw in this strategy, however — not all attributes used in entities have a role name. Thus the No-Null Rule would be violated. The entity class *Role* has been added along with a zero or one link with *AOIE*. *AOIE*'s key class is inherited by *Role*. With this architecture, an attribute used in an entity can have a role-name, and the information about the roles attributes play in entities is maintained.

5.4 Strengths and Weaknesses

Why IDEF1x? IDEF1x is a potent tool for data modeling. On the other hand, there are numerous other data modeling tools such as Chen's

ER [Chapter 6] and ENALIM [Chapter 4]. IDEF1x's strengths lie in its roots. Due to the strict standardization associated with Department Of Defense projects, IDEF1x should be saved from having numerous variants like ER. Having a standard is crucial to transfer of knowledge between organizations. It is hard enough to find time to learn more than one methodology, without having to learn variants of each.

IDEF1x also profits from its thorough description of the model development phases. The development process comes from IDEF1, and has better than ten years of testing behind it. The similarity of the model development processes between the two methods allows them to be easily used in combination (IDEF1 for information modeling and IDEF1x for conceptual data modeling). Without proper phases of development and interaction with experts and management, a modeling project is doomed no matter how strong the design of the underlying methodology.

Another strength of IDEF1x is its relationship to the other IDEFs. IDEF1x is part of a family of methodologies which form a framework for accomplishing a complete model of the enterprise. IDEF0 is used to model activities, IDEF1 for information, and IDEF1x for data. IDEF3 has recently become available for process flow and object-state-transition modeling and IDEF4 is available for object-oriented design.

A weakness of IDEF1x and nearly all methodologies is that the modeler must be experienced in order to create good models. Modeling is not an intuitive process, and many times models will have to be discarded due to a poor start. The simpler the methodology is to use the better, but the methodology must still have the necessary expressive power. A good example of a powerful concept which can be abused is the category relation. Whereas there are times when categories are necessary, there are others when they are used to create meaningless entities. Most inexperienced IDEF1x modelers tend to fall into the trap of using the categorization features of IDEF1x to model natural taxonomies as opposed to data taxonomies (as they were intended to be used). Because of the categorization components of the IDEF1x method many domain experts have fallen into the trap of trying to use the method for concept and terminology definition. Unfortunately the data modeling considerations that are built into the rules of IDEF1x do not allow it to function adequately for this purpose. The result is that much of the information gathered cannot be expressed or is expressed erroneously. For example, to function adequately as a language for concept and terminology definition, IDEF1x would have to be capable of expressing the fact that a SOW (statement of work) is a document and is a legal contract; or that a square is a polygon with four equal sides.

5.5 Integration With Other Methodologies

We have mentioned before that it is necessary to have more than one methodology if we hope to manage all the representational needs of a given enterprise. The sledgehammer approach just does not work. Having individual methodologies to capture each subset of the total enterprise representation requirements is not enough. The individual methodologies must work together as a cohesive and unified set. It is not possible at this time to expect a computer to draw all of the pertinent information from an activity model and create the information model. Whereas the computer can identify possible overlap, it is up to the modeler to define the overlap.

As an example of how the computer can identify overlap, let us look at IDEF0 [Chapter 3] and IDEF1x. In IDEF0, there are activities and concepts which are used as inputs, outputs, mechanisms, and controls of the activities. It is often suggested that these concepts could be automatically identified as entities in an IDEF1x model. While such integration cannot be completely automated, it can be eased with fairly simple tools. Say an IDEF0 model has been created. When the modeler moves on to the IDEF1x model, the concepts with their glossary text could be distributed among the source material log, the source data list, the entity pool, or the attribute pool to facilitate the generation of the model.

It is not enough to stop there though; there must be a conceptual schema through which data can be mapped back and forth between the models and a configuration management system for maintaining consistency between the models. If the concept is deleted from the IDEF0 model, it is likely that the respective entity will need to be deleted from the IDEF1x model. This is a rather simple case of integration called transliteration. Transliteration involves translating from the naming conventions of one methodology to those of another.

Unfortunately, there are few cases where there is a one-to-one mapping between model elements used in different methodologies. It may, however, be possible to develop production rules for translating specific configurations of model elements in one methodology into those of another methodology. For example, category relationships in IDEF1x map easily to zero-or-one links in IDEF1. Production rule translation between methodologies, however, is still limited.

The real magic begins with tagged inferences. Comparing the structures and textual descriptions of entities in different methodologies could infer a relationship between the entities. This is the meat of the integration issue and will require a great deal of research to obtain a solution. Once integration reaches this stage, the analysis portion of the modeling task will start to be automated.

If the scope of the integration is limited to integration with other data modeling methodologies, there are some specific issues which can be addressed. For instance, why do we need multiple methodologies for data modeling? Ideally, we do not need more than one. Unfortunately, the current methodologies have communities of modelers whose methodologies are the palettes they paint from. Where water colors and oil-based paints can both create a painting of a farm, the techniques used with the different mediums vary greatly. It is not reasonable to replace an artist's palette in the middle of his or her career. Thus, there will always be different methodologies with overlapping goals.

There are obvious similarities between the data modeling methodologies. For instance, there are entities in one form or another in all data modeling methodologies. Unfortunately, when it comes to constructs such as keys and relationships, there are major syntactic and semantic differences between the methodologies. For instance, how does the primary key from an IDEF1x model translate to attributes in an ER diagram?

These issues will continue to demand our attention. Fortunately, a number of efforts are currently under way to address methodology integration issues.

5.6 Conclusions

IDEF1x is a methodology for data modeling and conceptual schema design. Entities are defined by attributes and related to other entities. Keys are used to uniquely identify entities and pass information between entities. Categories of entities can be created which are discriminated by an attribute in each of the category entities.

Besides having a rigorous definition, IDEF1x also draws upon the other IDEFs to form a set of tools for modeling complete enterprises. Through integration at the methodological level, more sophisticated models will be able to be created and maintained. By looking at each methodology and understanding its semantic content, a better understanding of the integration issue can be developed. The path is a steep one, but does not appear to be insurmountable.

Appendix A. Abbreviations used in IDEF1x Metamodel

AN: Attribute Name; occurs in the key class of Attribute, uniquely identifies an attribute.

AOAD: Attribute Occurrence As Discriminator; an entity class.

AOIE: Attribute Occurrence In Entity; an entity class.

AOIK: Attribute Occurrence In Key; an entity class.

Attribute: an entity class.

CARD: attribute class containing information about the cardinality of the relation.

CAT-P: attribute class which identifies category relations.

DEN: Dependent Entity Name; attribute inherited from Entity. DEN specifies whether an entity the dependent entity in a relation.

EN: Entity Name; which uniquely identifies Entity.

Entity: an entity class.

IDENT-P: attribute class which specifies identifying relations.

IEN: Independent Entity Name;attribute inherited from Entity. IEN specifies whether an entity is the independent entity in a relation.

Key: an entity class which describes the characteristics of a key.

KID: Key IDentifier; occurs in the key class of Key, uniquely identifies a key.

KITR: Key Inherited Through Relation; an entity class.

OCC: OCCurrence number which distinguishes between similar attributes inherited from different entities.

OEN: Owned Entity Name; attribute inherited from Entity, which specifies the name of the entity where the attribute originated.

Relation: an entity class.

RID: Relation IDentifier; occurs in the key class of Relation, uniquely identifies a relation.

Role: an entity class.

Role-Name: attribute class of Role, describes the role an attribute plays.

SPEC-P: attribute class which identifies specific relations.

Status: attribute class which specifies whether a key is an alternate or primary key.

UEN: User Entity Name; occurs in the key class of AOIE .

References

ANSI/X3/SPARC, FDT...Bulletin of ACM - SIGMOD The Special Interest Group on Management of Data, ANSI/X3/SPARC, Study Group on Data Base Management Systems, Interim Report, 02/08, 1975.

ISO, Information processing systems - Concepts and Terminology for the Conceptual Schema and the Information Base, International Organization for Standardization, Technical Report 9007, 1987.

ISO, Concepts and Terminology for the Conceptual Schema and the Information Base edited by J.J. vanGriethuysen, March 15, 1982.

"IISS - Integrated Infomation Support System", D. Appleton Company, Inc, ICAM Project Priority 6201, Subcontract #013-078846, USAF Prime Contract #F33615-80-C-5155, Dec. 31, 1985.

Mayer, R.J. and the IDSE Research Team, "I^2SE^2 Report", Final Report, Knowledge Based Systems Laboratory, Texas A&M University, 1988.

6

Entity Relationship: Conceptual Schema Design

The IDSE Research Group is currently developing techniques to effectively integrate modeling methodologies. The approach in developing these techniques has been to analyze several modeling methodologies so that the factors that must be considered for integration could be identified. This chapter presents the Entity Relationship (ER) methodology and the analysis, in the form of an Integrated Computer-Aided Manufacturing (ICAM) DEFinition (IDEF) langage $IDEF_1$ model of the ER methodology.

6.1 History and Purpose

The Entity Relationship (ER) modeling methodology was originally developed in the mid-1970's by Dr. Peter Chen to aid in the design of database systems. The development of the ER approach was prompted by the recognition that the existing data models used for physical database layout design (e.g. the network model, the relational model, and the hierarchical model) were too "low level" for adequate modeling of structure and properties of a relational database and its mapping onto the "domain of discourse". As a result of this recognition, Chen first presented the ER approach in 1976. The ER model was intended to present a unified view of data, utilizing the advantages of the network, relational, and hierarchical models, while overcoming their individual disadvantages [Chen 76].

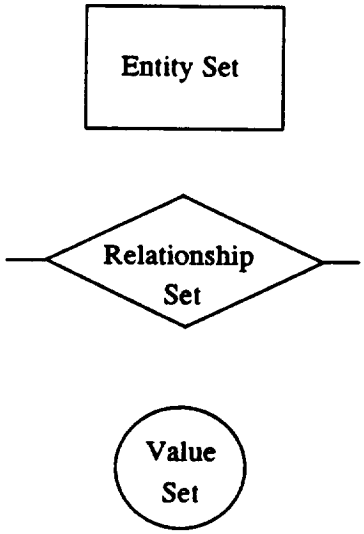
The ER model also appeared at a time when the concept of logical and physical views of data was in its infancy. Not too long after this method was introduced, the ANSI/X3/SPARC committee completed its three schema architecture for database system design. The components of the architecture are 1) the external schema, 2) the conceptual schema, and 3) the internal schema. The external schema describes the system as it appears to the user or application program while the internal schema specifies the physical database schema. The

envisioned purpose of the conceptual schema was to provide a basis for mapping the external schemas to the internal schemas. Chen envisioned the role of the ER method to encompass the specification of this conceptual schema.

The proported advantage of the intermediate conceptual schema was that this schema, once completed, would remain relatively constant over time, allowing isolation of the way the data was used (external schema) from the way it was physically stored (internal schema). The reason for this is that the conceptual schema presents an overall view of the information managed by a system. Changes in the internal and external schemas could take place without making any changes to the conceptual model. The ER approach was proposed for the development of the conceptual schema by taking a "real world" approach toward describing the system. This description would be independent of any user or database manager view of the system, providing a stable base upon which to develop the external and internal schemas.

6.2 Syntax and Semantics

In an ER model of the real world, things are recognized as either entities or relationships among entities. An entity is just some "thing" that exists within the system being modeled. Entities that share common characteristics are grouped into entity sets. A relationship shows some interaction between entities taken from one or more entity sets. Relationships that relate entities from the same entity sets and describe the same interaction are grouped into relationship sets. More formally, a relationship r, an element of the relationship set R which is defined on entity sets E_1, E_2, ..., E_m, is expressed as a tuple $r = (e_1, e_2, ..., e_m)$, with the meaning that entities e_1, e_2, ..., e_m are mutually related with respect to R [Sakai 83].



ER Syntactic Elements

To more completely define these objects, attributes can be defined for both entity sets and relationship sets. An attribute is a function that maps a particular entity or relationship onto a certain value that is a member of a value set or Cartesian Product of value sets. A value set simply indicates the type of value that a particular attribute may have. The definition of these value sets is also required when defining an attribute for an entity set or relationship set. It is possible for different attributes to map to the same value set.

An Entity Relationship diagram uses four syntactic elements to represent the entity sets, relationship sets, attributes, and value sets (see insert). A rectangular box is used to denote an entity set and the name of the entity set is placed inside the box. To indicate a relationship set, a diamond shaped box with lines running from the relationship set to the related entity sets is used. As with the entity set, the

name of the relationship set is placed within the diamond shaped box. To represent a value set, a circle, with the name of the value set inside, is used. An attribute of an entity set or relationship set is specified by drawing an arrow from the entity set or relationship set which is described by the attribute to the appropriate value set for that attribute. A label next to the arrow gives the name of that attribute. When an attribute maps an entity or relationship set instance onto a Cartesian Product of value sets, a split arrow is used to link the entity or relationship set with the value sets in the Cartesian Product.

Figure 6.1 shows a simple diagram modeling a typical office situation. Again, the rectangular boxes represent the entity sets (labeled Employee, Project, and Department). The example in Figure 6.1 also defines two relationship sets using diamond shaped boxes (labeled Worker and Has/In). Also, note that the links (represented by lines) connecting the relationship set to entity sets are annotated at the ends. The cardinality of a relationship is described using these annotations. For instance, the Has/In relationship set, relating Department entities to Employee entities, has a one to many cardinality (sometimes written 1:n). The reading of the relationship denoted by the "has" relationship set is "a Department entity can have n (an arbitrary number of) Employees." The reading of the relationship denoted by the "In" relationship set is "an Employee can be in only one Department." An ER diagram also allows one to one (1:1) and many to many (m:n) relationships [Chen 77].

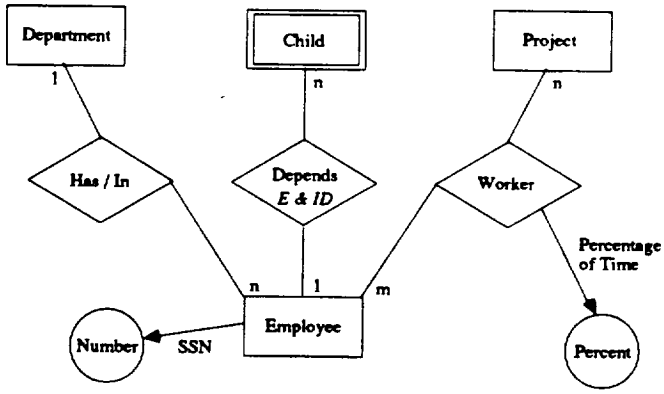


Figure 6.1 Example ER Diagram

Figure 6.1 also shows the definition of attributes and value sets. Notice that both entity sets and relationship sets can be used to display associated attributes. SSN is an attribute for the Employee entity set and Percentage of Time is an attribute for the Project-Worker relationship. In the diagram, a labeled circle represents a value set.

The example in Figure 6.1 also describes additional features of the ER method. Notice the double box surrounding the Child entity set

and the E & ID describing the Depends relationship set. These descriptive features illustrate the "Existence" and "Identification" Dependencies. [Chen 77]. In this case, the Child entity set is both existent dependent and ID dependent on the Employee entity set. Both dependencies occur through the Depends relationship set. Existence dependency tells us that a child entity cannot exist within this system unless the employee entity upon which the child depends also exists. Similarly, ID dependency tells us that identification of a child entity depends on the ID of the employee entity upon which the child depends. This example presents a situation where an entity set is both existence and ID dependent on another entity set. It is not required that this always be true. It is possible for an entity set not to be ID independent and yet still be existence dependent and vice versa. In either of these two cases, the double box still surrounds the dependent entity set, but only the E or ID will appear in the diamond for that relationship set.

An additional point not obvious in this example is the notion of identification. A primary key is a collection of attributes that will uniquely identify an occurrence of an entity set. In building an ER model, the attributes making up the primary key must be identified. Occurrences of relationship sets also have a primary key to identify them. The difference is that the key of a relationship set is determined by combining the keys of the entity sets related by the relationship set. As such, the relationship set does not really have a key of its own since the key is derived. The significance of this fact becomes more evident in the discussion of attributes in the Metamodel section of the paper.

At this point, it should be noted that different versions of ER diagrams exist. In fact, the different versions make up a spectrum of ER types. On one end, there is the ER model originally suggested by Chen that allows n-ary relationships (relationships defined on more than two entity sets) and attributes defined for both entity sets and relationship sets. On the other end, there is the version that allows only binary relationships between entity sets and does not allow for the definition of any attributes [Chen 81]. Each version along this spectrum was developed to overcome a certain disadvantage of existing versions or to provide the capability for a certain situation that could not be handled with current versions. This chapter is concerned only with the ER method originally introduced by Dr. Chen.

6.3 Metamodel

This section discusses an information model of the Entity Relationship methodology. This model is represented in IDEF$_1$. In discussing this model, careful attention must be given to the fact that both ER

and IDEF$_1$ have similar terms for objects in their respective models. In the following discussion, it is important to remember that entity sets and attributes are part of ER models while entity classes and attribute classes are part of IDEF$_1$ models. The following discussion will attempt to prevent any confusion in terminology.

The complete IDEF$_1$ metamodel of the Entity Relationship method is given in Figure 6.2. For discussion purposes, the metamodel will be broken up into logical units and each unit will be discussed individually. This should make the explanation of the metamodel more understandable.

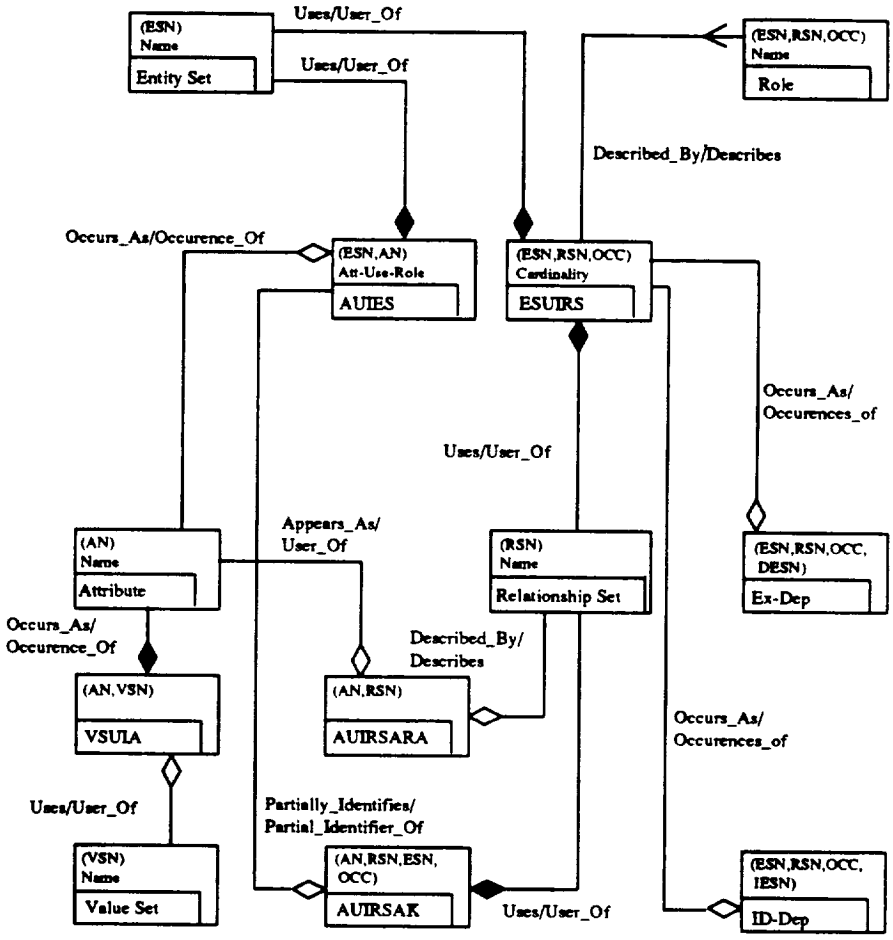


Figure 6.2 Entity Relationship in IDEF$_1$

6.3.1 Base Entity Classes

In the metamodel (see Figure 6.2) there is a corresponding entity class for each of the four ER objects (entity set, relationship set, attribute, and value set) that occur in an Entity Relationship model. Each of these entity classes captures the information maintained by each of

these objects in an ER model. The *Entity Set* entity class has an Entity Set Number (ESN) attribute as part of its key class that will uniquely identify that Entity Set. This entity class also has a Name attribute that captures the name of the Entity Set. The other three base entity classes have similar key and attribute classes. The *Relationship Set* entity class has a Relationship Set Number (RSN) in its key class and a Name attribute class. The *Attribute* entity set has an Attribute Number (AN) in its key and a Name attribute. Finally, the *Value Set* entity class has a Value Set Number (VSN) in its key class and a Name attribute class. Besides these four entity classes, additional entity classes have been added to show information that the ER model maintains about the interaction between these four entity classes and to show special relationships that exist between the entity classes.

6.3.2 Entity Set/Relationship Set Interaction

Figure 6.3 shows the interaction between the Entity Set and Relationship Set entity classes. A relationship set may relate one or more entity sets and an entity set may be involved in one or more relationship sets. The *Entity Set Use in Relationship Set* (ESUIRS) entity class reflects this many to many situation. The ESUIRS entity class inherits the ESN attribute class from Entity Set and the RSN attribute class from Relationship Set into its key class. The additional OCC

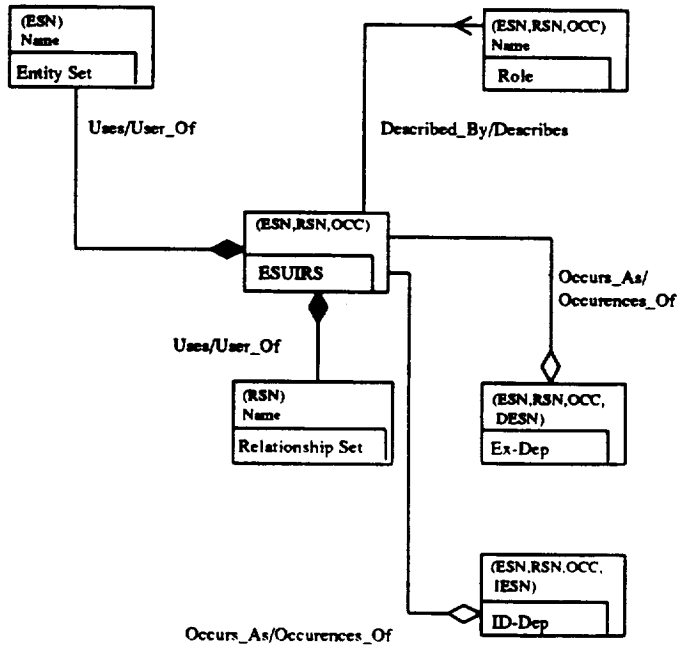


Figure 6.3 Entity Set - Relationship Set

attribute class completing the key class makes the distinction between multiple occurrences of the same entity set in the a relationship set. This OCC attribute ensures that a relationship will have a unique identification.

Another entity class in Figure 6.3 represents the *Role* of an entity in a relationship. Whenever an entity set is used in a relationship, the role that the entity plays in the relationship can be specified. Notice that a role is not always required when relating entity sets. An example of this situation might be a marriage relationship defined between two entities from the Person entity set. Since marriage is a binary relationship defined on the same entity set (Person), additional information must be maintained to make the distinction between the two entities. In this example, one entity would be given the husband role while the other entity would be given the wife role to further describe the two entities. This entity class inherits its key class directly from ESUIRS and requires no additional attributes in its key class. This is true as the relation is uniquely identified by the OCC attribute class. However, the Name attribute has been added to represent the name of the role the entity is to assume in the relationship.

The final two entity classes in Figure 6.3, *Ex-Dep* and *ID-Dep* represent the existence dependency and the identification dependency of an entity involved in a relationship. Their structure within the model is almost identical. The ESUIRS entity class has a one to zero, one, or many link with the Ex-Dep entity class. This indicates that one entity set involved in a relationship can be existent dependent on one or more other entity sets involved in the relationship. The ESUIRS entity class also has the same type of links with the ID-Dep entity class. Again, this is saying that one entity involved in a relationship can be identification dependent on one or more other entities involved in the relationship. Both entity classes inherit their key classes from ESUIRS. But, knowing that an entity set is dependent, without knowing on which entity set it depends, is not very useful. As a result, the Dependent Entity Set Number (DESN) attribute class was added to the key class of Ex-Dep and the Independent Entity Set Number (IESN) attribute class was added to the key class of ID-Dep. In each case, the attribute identifies the entity set upon which the dependent entity set depends.

6.3.3 Entity Set/Relationship Set/Attribute Interaction

Figure 6.4 outlines another portion of the metamodel. This portion represents the interaction between the Entity Set and Relationship Set entity classes and the Attribute entity class. As was mentioned before, both entity sets and relationship sets can have attributes. But, again,

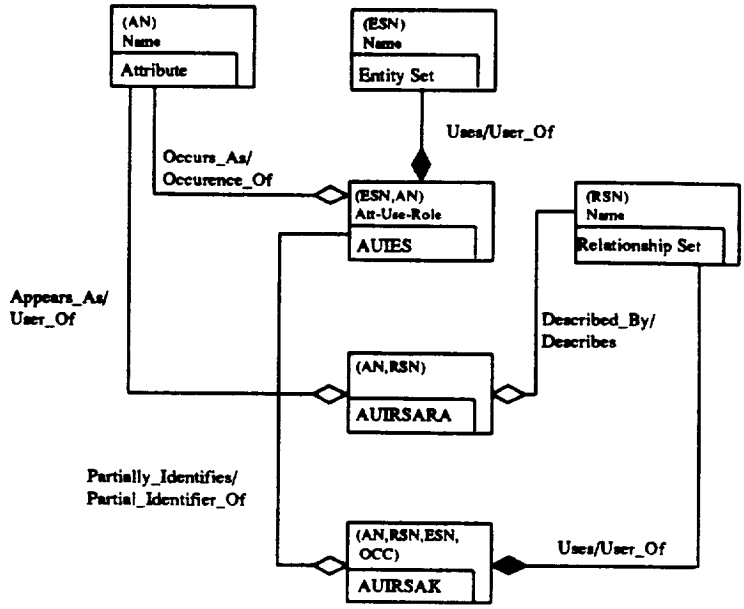


Figure 6.4 Attribute-Entity Set-Relationship Set

a many to many situation exists as an attribute can apply to many entity or relationship sets while an entity set or relationship can have many attributes. For each case, to correctly capture this information, an additional entity class was defined. The *Attribute Use in Entity Set* (AUIES) entity class was added to capture the multiple use of attributes by many entity sets and to capture the possession of multiple attributes by an entity set. The key class of this entity set is made up of the ESN of the entity set being described and the AN of the attribute. In addition, The Att-Use-Role attribute of AUIES indicates whether the attribute is being used as part of the key of the entity set or whether it is just a descriptive attribute.

The *Attribute Use in Relationship Set as Relationship Attribute* (AUIRSARA) was added to resolve the many to many situation between Attribute and Relationship Set in the same way that AUIES resolved the problem for Entity Set and Attribute. Similarly, this entity class inherits its key class from Relationship Set and Attribute. But, notice that an Att-Use-Role attribute does not appear in this entity class. This is because of a distinction in ER models between the use of attributes for describing relationship sets and the use of attributes for identifying relationship sets. Remember from Section 6.2 that a relationship set derives its key from the entity sets that the relationship set relates. This derivation of the primary key must be reflected in the metamodel. When an entity set is involved in a relationship, the record that a transfer of an attribute in the key of the

entity set to the key of the relationship set has occurred is maintained through the *Attribute Use in Relationship Set as Key* (AUIRSAK) entity set. It is in the two entity classes, AUIRSARA and AUIRSAK, that the distinction between a descriptive attribute and an identifying attribute of a relationship set is maintained.

AUIRSAK inherits part of its key, the AN and ESN attribute classes, from the AUIES entity class since any attribute in the key of a related entity set will also be an attribute in the key of the relationship set. The RSN migrates to the key class from the Relationship Set that relates the entity set. And finally, the OCC attribute is necessary to distinguish between multiple occurrences of the same entity set in a relationship set.

6.3.4 Attribute/Value Set Interaction

Finally, Figure 6.5 shows the interaction of the Attribute and Value Set entity classes. Again, a many to many situation exists between the two classes. An attribute can be used to describe many entity and relationship sets but, every time, map to a different value set. On the other hand, a value set can be used as the range for many attribute functions. As a result, the *Value Set Use in Attribute* (VSUIA) entity class was added to capture these situations. The key class of VSUIA consists of the AN from the attribute being defined and VSN from the limiting Value Set.

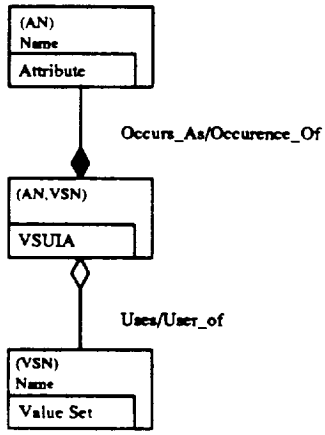


Figure 6.5 Attribute - Value Set

6.4 Tips and Traps

As mentioned previously, ER models have proven very useful in developing conceptual schema for database systems. The following guidelines should assist in the development of ER models:

1. Identify the entity sets.
2. Identify the relationship sets.
3. Draw the ER model with the entity and relationship sets.
4. Identify attributes and value sets.

6.5 Strengths and Weaknesses

As has been mentioned, the greatest strength of the ER method is its ability to effectively represent the conceptual schema (as used in the context of database management systems), since it was originally developed as a mechanism for logical database design. ER's effective manner of representing the conceptual schema is derived from the fact that it produces relatively simple and intuitive descriptions of the systems being modeled, and from the fact that effective techniques have been developed to translate a completed ER model into an equivalent data structure definition [Chen 77]. This allows an easy development of the internal schema from the conceptual schema represented in an ER model.

However, notice that these strengths all depend on the existence of a completed ER model. A completed ER model is easy to understand and easy to translate. But, there is no easy way to produce this ER model. Section 6.4 outlines a step by step process to follow when producing an model. But, for example, just how does a modeler go about identifying the entity sets that will exist within the model? "There is no evidence to suggest that it is easy or natural to select, a priori, the entities, attributes, and relationships for ER conceptual schema. On the contrary, the opposite seems to be true: the task is commonly regarded as subjective, difficult, and iterative" [Nijssen 88]. We believe that the primary reason for this difficulty is that ER is a design method. It is intended to assist a designer in organizing, communicating, and analyzing his/her design. The problems with its use arise when non-designers attempt to use it to model concepts and terminology in their domain or when programmers attempt to use it to model internal data structure. The first of these applications is more appropriately a task for the ENALIM method. The second is the design of a data charting technique. Problems experienced during a misuse of a method (application beyond its design limits) should not be considered a valid basis for criticism of that method. However it can be said that to increase the usefulness and effectiveness of the ER method for database designers, qualitative techniques and decision procedures for identifying and defining the entity sets, relationships, attributes, and value sets must be developed.

6.6 Integration With Other Methodologies

Integration of modeling methodologies can take two approaches. The first approach is to integrate a methodology with another methodology that is very similar or that is used for the same purpose. The advantage of this is twofold. First of all, this integration would allow people familiar with different methods to understand models, originally produced in another methodology, that have been translated into the methodology they are familiar with. This ability might promote joint efforts in developing models, even when the modelers use two different, but similar methodologies. Chen also points out that the integration of similar methodologies would also allow the equivalence of two methods to be proven [Chen 81].

The second approach is to integrate methodologies that are not necessarily similar but, when used together, provide a robust means of developing effective information models. This integration will not involve a translation from one method's syntax to another method's syntax. Instead, this will require the identification of common elements within the methodologies themselves so that the equivalent portions of models in the two or more methodologies can be effectively integrated.

6.7 Conclusions

The purpose of this chapter has been to describe the IDSE Research Group's analysis of the ER approach and the development of the ER metamodel (a model of ER in $IDEF_1$). This metamodel is the primary basis from which integration decisions concerning ER will be derived. By developing a metamodel for ER, we hope to generate a very accurate description of the methodology. The advantage of the metamodel is two-fold. First of all, common elements of different methodologies can be more easily identified. In addition, integration problems can be more easily resolved by having a more complete and less ambiguous understanding of the different methodologies. The metamodel provides a common reference from which decisions can be made.

In performing this research, it is the goal of this group to develop techniques that will allow the integration of multiple modeling methodologies. The benefit of integrating these methodologies will be the generation of more complete system models. Each method, on its own, is used for a special purpose or used to represent a certain perspective of the system being modeled. By integrating these methods, a more global perspective and a more complete model of the system will result.

Appendix A. Abbreviations used in the Entity Relationship Metamodel

Attribute: an entity class.

Att-Use-Role: attribute in attribute class of AUIES which differentiates between attributes that occur in key classes and descriptive attributes.

AN: Attribute Name; occurs in the key class of Attribute, uniquely identifies an attribute.

AUIES: Attribute Use In Entity Set; an entity class.

AUIRSARA: Attribute Use In Relationship Set As Relationship Attribute; an entity class.

AUIRSAK: Attribute Use In Relationship As Key; an entity class.

Entity Set: an entity class.

DESC: Dependent Entity Set Number; occurs in the key class of Ex-Dep, uniquely identifies an existence dependency.

ESN: Entity Set Number; occurs in the key class of Entity Set, uniquely identifies an entity set.

ESUIRS: Entity Set Use In Relationship Set; an entity class.

Ex-Dep: Existence Dependency; an entity class.

ID-Dep: IDentification Dependency; an entity class.

IESN: Independent Entity Set Number; occurs in the key class of ID-Dep, uniquely identifies an identification dependency.

Name: attribute which captures the name of the Entity Set, Relationship Set, Attribute, Value Set or Role, according to the context in which it is used.

OCC: Occurrence number which distinguishes between multiple occurrences of the same entity set in a relationship set. It ensures that a relationship will have a unique identity.

Relationship Set: an entity class.

RSN: Relationship Set Number; occurs in the key class of Relationship Set, uniquely identifies a relationship set.

Role: an entity class.

Value Set: an entity class.

VSN: Value Set Number; occurs in the key class of Value Set, uniquely identifies a value set.

VSUIA: Value Set Use In Attribute; an entity class.

References

Chen, Peter P.S. "ER — A Historical Perspective and Future Directions", in: Davis, C. G. et al (ed.), Entity-Relationship Approach to Software Engineering, (North-Holland, Amsterdam, 1983).

Chen, Peter P.S. "Framework for E-R Models," in: Chen, P.P. (ed.), Entity-Relationship Approach to Information Modeling and Analysis, (ER Institute, Saugus, CA, 1981).

Chen, Peter P.S. The Entity Relationship Approach to Logical Database Design, QED Information Sciences, 1977.

Chen, Peter P.S. "The Entity-Relationship Model — Toward a Unified View of Data," ACM Transactions on Database Systems., Vol. 1, No. 1, (March 1976), pages 9-36.

Nijssen, G. M., Duke, D. J., and Twine, S. M. "The Entity-Relationship Data Model Considered Harmful," Effective Relational Database Design, (Digital Consulting Inc., Sydney, 1988).

Sakai, H. "A Method for Entity-Relationship Behavior Modeling," in: Davis, C. G. et al (ed.), Entity-Relationship Approach to Software Engineering, (North-Holland, Amsterdam, 1983).

7

Data Flow Diagrams: Design and Analysis[1]

The Data Flow Diagram (DFD) methodology is a widely used modeling methodology in which the modeler focuses on representing a system from the viewpoint of the data in the system. Its primary application is in design and analysis. The first and second sections of this chapter briefly describe the background, the syntax and semantics of Data Flow Diagrams. The third section presents an $IDEF_1$ information model of the DFD methodology. This information model represents the structure of information needed to support the functions of a system or operating environment. The fourth and fifth sections examine the strengths and weaknesses of the DFD technique and common tips and traps encountered when using DFDs. Finally, the issue of integration with other methodologies is examined and concluding remarks are presented.

7.1 History and Purpose

Concepts similar to those used in data flow diagrams (DFD) have actually been used since the 1940's in flowcharts and Petri Networks. Since then, the concept of representing the flow of data through a system has been used for modeling mathematical systems in 1973 [Whitehouse 73], for structured program design in 1975 [Yourdon 75], and for systems analysis in 1977 [Ross 77]. DFDs are primarily used for understanding and working with a system of any complexity at the logical level [Gane 77]. They allow a system to be decomposed into a network representation describing each component and the manner in which each relates to the other components. DFDs force the modeler to present the system from the viewpoint of

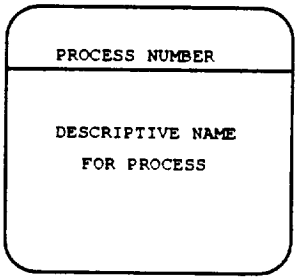
[1] Funded in part by Tandem Computers Incorporated

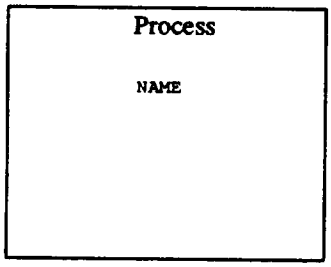
the data. This view can often be in sharp contrast to the viewpoint of an individual or a group of individuals. In classical analysis, the user's viewpoint and the system's viewpoint were the primary focus. This difference is significant when examining an overall picture of the system. By concentrating on the data, a larger view of the system can be grasped instead of the smaller more biased views that different managers, divisions, machines, and other data processors are confined to from a person, organization, or system point of view.

7.2 Syntax and Semantics

Data Flow Diagrams are composed of four basic elements: processes, external entities, data stores, and data flows. Notationally, there are several variations for drawing DFDs. Throughout this paper, the convention used by Whitten, Bentley, and Ho [Whitten 86] will be used. Appendix A shows several of the most common notational conventions. The following section describes the syntactic and semantic rules of data flow diagrams.

7.2.1 Process





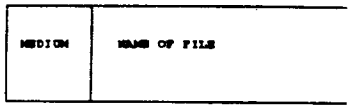
A DFD "process" represents the transformation of incoming data flow(s) into outgoing data flow(s) [DeMarco 79]. It represents some type of work performed on data and is required to have a descriptive name. Notational conventions for representing DFD processes include rounded rectangles, circles (bubbles), ovals and square boxes. In a complete DFD, each process will be assigned a unique reference number. A process must be either at the source and/or destination end of a data flow. Valid combinations, therefore, would be a process connected to another process, a process connected to a data store, or a process connected to an external entity. A data store represents the existence of a temporary storage place for data. An external entity represents a boundary which lies outside the context of a system. Both data stores and external entities will be discussed in more detail later in this section.

The transformation that a process performs on data includes moving or routing data, performing computations, splitting data into subsets, combining data from different sources, and changing the basic structure of the data. Data might undergo sorting, verification, formatting, or other similar operations in a transformation process.

7.2.2 External Entity

External Entity

A DFD "external entity" (EE) represents a boundary which lies outside the context of a system. An EE denotes the system's connec-



Data Store

tion to the outside world. It can be a person or an organization but must be either an originator or receiver of system data [DeMarco 79]. EEs are also called data sources and sinks. Data may flow both to and from an external entity. Notationally, square boxes are typically used to represent external entities. Although an external entity may be connected to a process, it may not be connected to another external entity or a data store.

7.2.3 Data Store

Data Flow

A DFD "data store" (DS) represents the existence of a temporary storage place for data. Basically, DSs represent collections of data used and maintained by the system being modeled. Examples include tapes, files, databases, in/out boxes, and books. Notational conventions include open ended rectangles with an optional slot for the medium and straight lines. Both notations require a DS name. DSs are also referred to as files. A data store may be connected to a process but it may not be connected to another data store or an external entity.

7.2.4 Data Flow

A DFD "data flow" (DF) represents the existence of a transfer of packets or parcels of information of known composition [DeMarco 79]. DFs depict reports, documents, computer input, memos, and any other information flow. They are represented by a line with an arrowhead at the destination end. Data flows must begin and/or end at a process and must either initiate a process or result from a process. They may converge and diverge.

Diverging data flows have a single source and multiple destinations. A common example would be a purchase order in which the order comes from a single source, the sales department. When duplicate copies of the purchase order are distributed to different departments such as accounts receivable and shipping, the purchase order or data diverges. Note that each duplicate copy is a packet of known composition as required.

Converging data flows have multiple sources and a single destination. Several distinct documents from different departments might converge to form a single combined document. A student's college transcript would be one example. In general, diverging data flows are more common that converging data flows.

7.2.5 Differences between DFDs and Flow Charts

There are several important distinctions between data flow diagrams and flow charts [Whitten 86]. One difference is that DFD processes can describe parallel operations while flowcharts generally only show sequential processes. A second difference is that DFDs show the flow of data through a system while flowcharts explicitly show looping and decision constructs. A third difference is that DFDs can show timing differences between processes. Finally, while flow charts have a clearly defined starting point, DFDs do not have this requirement.

7.2.6 Differences between DFDs and Logical DFDs

Another area where possible confusion might occur is in differentiating between data flow diagrams and logical data flow diagrams (LDFDs) [Whitten 86]. Basically, LDFDs avoid implementation details by showing only the essential features of the system. They are used to specify the logical system requirements.

DFD data flow and data store names describe implementation details while LDFD data flow and data store names describe the data contained and avoid how the data is stored or implemented. DFD allow several types of processes that, due to their implementation-dependent nature, are not necessary in LDFD. Examples would be: 1) processes that do not change the composition or nature of incoming data flows, and 2) processes that would not be necessary if the system were implemented differently. DFDs often have processes that include multiple tasks performed without any real data flow. LDFDs would break these processes into several processes each performing an individual task. More consolidation of duplicate processes is done in LDFDs than in DFDs. LDFDs tend to restructure the sequence of processes to capitalize on parallel processing when possible. LDFD data stores should be consolidated to minimize redundant data storage. LDFDs attempt to eliminate bias of how things are done by only representing the necessary requirements.

LDFDs are often used by domain experts to describe how a current system works. They are also used to convey ideas about how a new system might work. DFDs are most often used as design specifications by a programming team.

7.3 Metamodel

The metamodel of the data flow diagram methodology was developed using $IDEF_1$. In this paper, the DFD methodology is being treated as a system and the information managed by this method is being modeled using $IDEF_1$. The information and relationships dis-

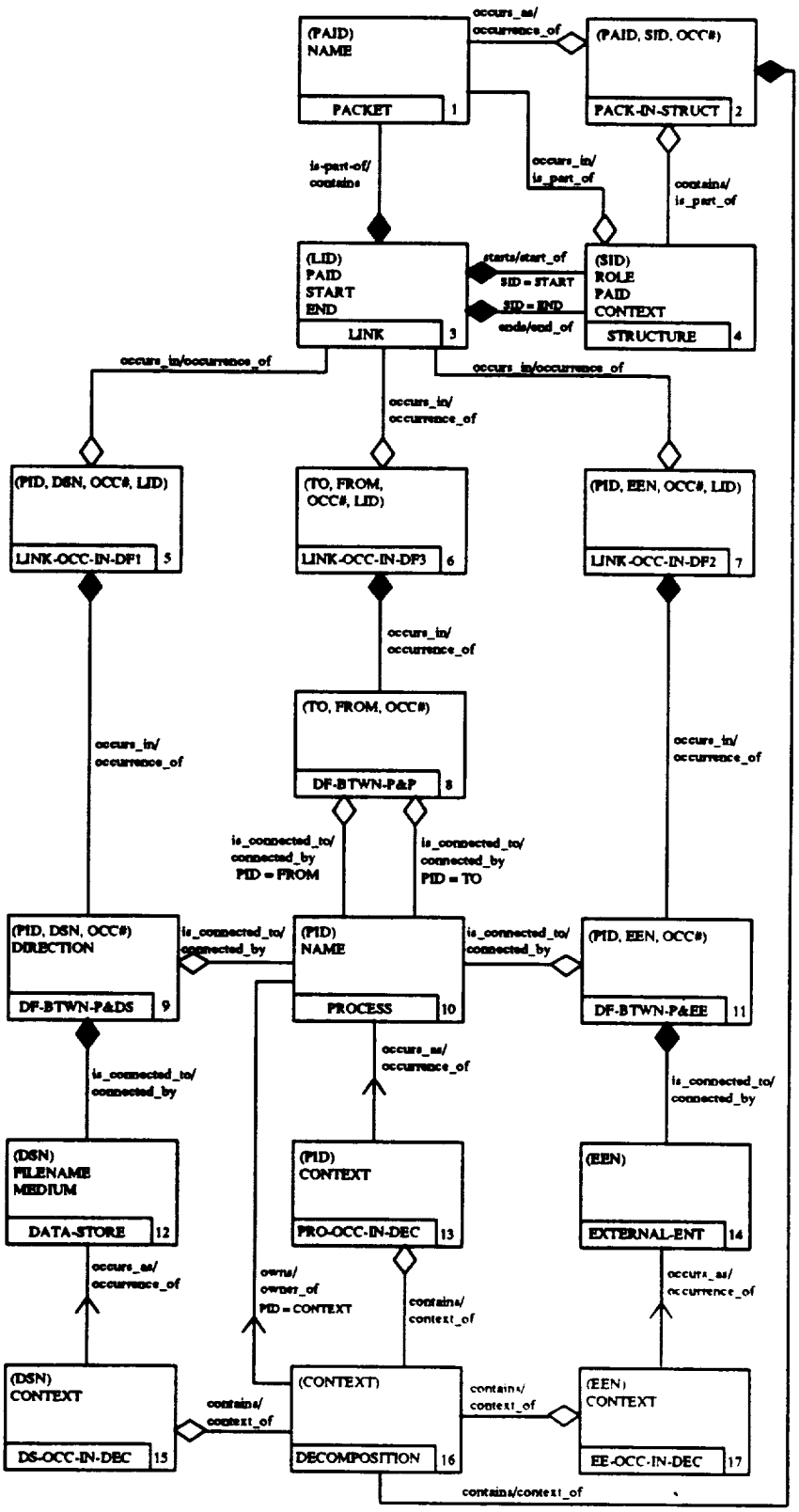


Figure 7.1 IDEF$_1$ Metamodel of DFD

played in the DFD metamodel provide a view of the DFD method system from the viewpoint of the information managed by that method.

The objective, when developing the DFD metamodel, was to model the information that the DFD method maintains about processes, data stores, data flows, and external entities. By modeling the information kept by each DFD model element, their relationships could be studied to find commonality across system engineering methodologies. For example, a DFD process and an IDEF0 [SofTech 81] activity actually maintain the same information although they have different model element names. This commonality could be useful when trying to build a DFD view of a system given an IDEF0 model and view.

A key to achieving this objective was to first develop a way to model the DFD data flow. The single DFD data flow actually turns out to represent several different "types" of data flows based on the information maintained by the data flow as it relates to different entity classes. One alternative to modeling the "different" data flow "types" in this manner would be to use a constraint language to specify legal and illegal constructs. The research team has developed a constraint language based on first order logic and basic set theory. It will provide a neutral representation language to give the methodologies more expressive power. This will enable the modeler to avoid awkward constructions and create simpler models since the constraint language can be used to handle unusual relationships and cases.

The four key concepts needed to understand the structure of the DFD metamodel are:

- differences between processes
- the relationships between processes
- decomposition or leveling
- relationship of the structure entity class to converging and diverging data flows

These four concepts will be covered in the next three sections.

7.3.1 Relationship of process to other entity classes

DFDs explicitly have only a single data flow type; however, it is used in several different ways. Thus, the DFD metamodel has three distinct entity classes to capture the different information inherent in each use. This view was taken in the DFD metamodel resulting in the entity classes: Data Flow BeTWeeN Process and Data Store (DF-BTWN-P&DS), Data Flow BeTWeeN Process and External Entity (DF-BTWN-P&EE), and Data Flow BeTWeeN Process and Process

(DF-BTWN-P&P). This structure models the information maintained by data flows associated with a process and a data store pair, a process and an external entity pair, and between two processes, respectively.

Why not simply model the single data flow? The problem arises when trying to properly inherit attributes. Processes, data stores, and external entities do not maintain the same attributes. A constraint list would also be necessary to prevent illegal combinations of processes, data stores, and external entities. For example, a data flow connecting two processes maintains information about a destination process (*TO*), a source process (*FROM*), and an occurrence number (*OCC#*). A data flow connecting a process and a data store maintains a process id (*PID*), a data store name (*DSN*) and an occurrence number. This becomes significant when *LINK-OCC-IN-DF3* inherits *TO*, *FROM*, and *OCC#* from *DATA-FLOW-3* and then *LINK-OCC-IN-DF1* inherits *PID*, *DSN*, and *OCC#* from *DATA-FLOW-1*.

A process is the independent entity in each of the weak many to one relationships with the data flow entities (DF-BTWN-P&DS, DF-BTWN-P&EE, DF-BTWN-P&P). Figure 7.2 highlights these relationships and the procss entity class relationship to the decomposition entity class as depicted in the metamodel. A more detailed discussion of the decomposition entity class will be presented later (see Section 7.3.2).

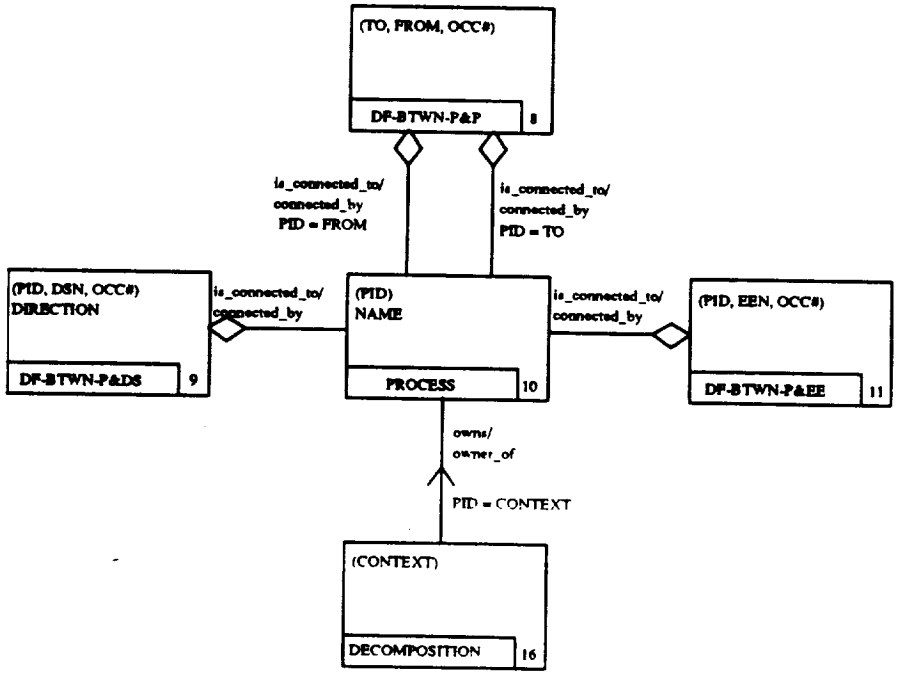


Figure 7.2 Decomposition, Process and Data Flows

Figure 7.3 highlights another portion of the metamodel. The DF-BTWN-P&DS entity class models the information that a DFD maintains about a data flow between a process and a data store. A data store is the independent entity class in the strong one to many relationship to dependent DF-BTWN-P&DS. The DF-BTWN-P&DS key class is composed of a process id (*PID* - inherited from process), data store name (*DSN* - inherited from data store), and an occurrence number (*OCC#*). Its attribute classes include the attribute class direction which maintains the source-to-destination direction of the data flow. In other words, when connecting a process and a data store, the fact that the flow of data is from process to data store or the reverse is maintained.

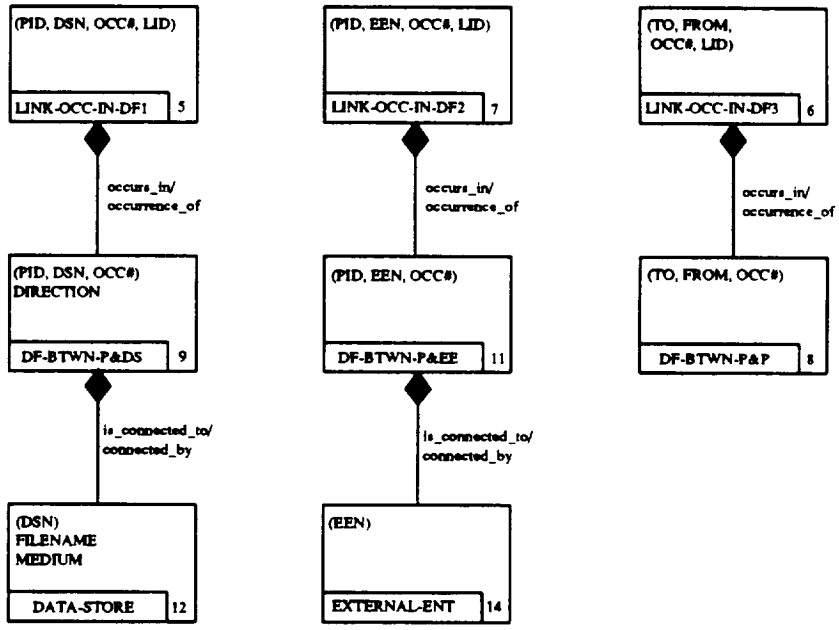


Figure 7.3 Data Flow Entity Classes

DF-BTWN-P&EE models the information that a DFD maintains about a data flow between a process and an external entity. The "external entity" is an independent entity in a strong one to many relationship to DF-BTWN-P&EE. In other words, for every external entity, there must be one or more DF-BTWN-P&EE. The DF-BTWN-P&EE key class and attribute class are identical to that of DF-BTWN-P&DS with one exception. Since DF-BTWN-P&EE connects a process to an external entity, the DSN (data store name) is replaced with *EEN* (external entity name).

DF-BTWN-P&P models the information that a DFD maintains about a data flow between two processes. The attribute class direction is not necessary in this case due to the construction of the key class. The key class for DF-BTWN-P&P consists of the destination process

(*TO*), the source process (*FROM*), and the occurrence number (*OCC#*).

7.3.2 Leveling/decomposition

Leveling in data flow diagrams allows a process to be described in finer detail on another level of the diagram. This mechanism allows various levels of abstraction. The top level of a DFD might merely show a very general view of the system while the lowest levels will show the most detail. A process that at one level shows a document transformed in one department may at a second level show which offices in the department are involved. This detail might not be desirable at the top level but might be necessary to model the system appropriately.

The leveling or decomposition aspect of the DFD metamodel is shown in Figure 7. 2. A process is the independent entity class in the zero or one to one relationship to the dependent decomposition entity class which inherits the key class of process, process ID (*PID*). This relationship is provided, as previously described, for modeling the levels of a data flow diagram. A decomposition is the independent entity class in the weak many to one relationship to a dependent occurrence of a process (*PRO-OCC-IN-DEC*), external entity (*EE-OCC-IN-DEC*), and/or a data store (*DS-OCC-IN-DEC*). Each occurrence has a one to zero or one relationship to its respective entity class (*DATA-STORE*,*PROCESS*, or *EXTERNAL-ENT*) and inherits its key class from that class.

7.3.3 Role of the Structure and Link Entity Classes

The data flow was the most difficult construct in the DFD methodology for which to model the information structure. As noted before, data flows depict information flow and there must be an associated information "packet" of known composition. The simplest case is a single data flow from one process to another. The most complex case occurs with converging or diverging data flows. The difficulty arose when trying to model the flow of information as it split into multiple data flows (diverged) or merged into a single data flow from several data flows (converged).

The *STRUCTURE* entity class was created to model the splits (diverges) and joins (converges) that can occur with data flows. For our modeling purposes, the data flow is viewed as being broken up into pieces called structures and links. Collectively, these pieces make up a path. A path, therefore, is modeled as a combination of structures and links. Please refer to Figure 7.4.

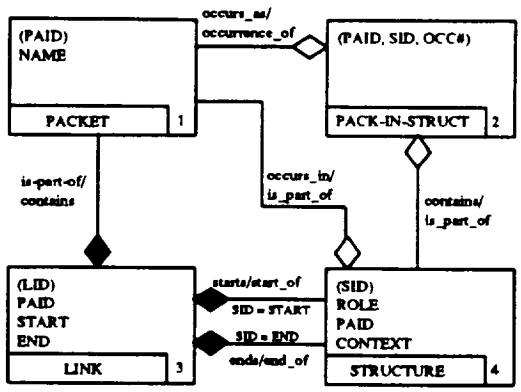


Figure 7.4 Structure,Link,and Packet Entities

A structure represents a connection point on the path between the source and destination. It occurs at any point in which a data flow starts or ends and wherever data flows diverge or converge. Links represent the part of the path between the structures. A simple path from process A to process B would consist of a structure at process A, a structure at process B, and a link between the two structures. A path containing process A and a diverging data flow which ends at process B and process C would be slightly more complicated. It would contain an additional structure at the split, structures at each process as before, and a link from process A to the diverging structure, and links from the diverging structure to the structures at processes B and C.

Therefore, in the DFD metamodel a link is the independent entity in a weak one to many relationship to *LINK-OCC-IN-DF1*, *LINK-OCC-IN-DF2*, and *LINK-OCC-IN-DF3*. *LID* is inherited in each of these cases from *link*'s key class. Note that *link*'s descriptive attribute classes consist of a packet id (*PAID*), a source (*START*), and a destination (*END*).

A *STRUCTURE* entity is the dependent entity in a weak one to many relationship to *PACKET*. It is the independent entity in a weak one to many relationship to *PACK-IN-STRUCT*. It is also the independent entity in two strong one to many relationships with link.

7.4 Strengths and Weaknesses

The characteristics of a methodology that are viewed as strengths or weaknesses are generally subject to opinion and this section is no exception in that regard.

The DFD modeling approach focuses the modeler's viewpoint to reflect that of the data processors. This is one of its major strengths

and allows the modeler to avoid the bias associated with a person or organization's local view and obtain a more global view. Another strength is the fact that the DFD methodology encourages the modeler to decompose and partition information into smaller units so that the higher levels of the diagram are more abstract and the lower levels show more detail. This top-down approach is valuable in analysis and in understanding complex models. DFD processes can also operate simultaneously which is a key advantage over techniques that only allow sequential processes.

There is some confusion concerning the difference between physical DFD (PDFD) and logical DFD (LDFD) models [Whitten 86]. This can be a strength when used correctly, but more often is a weakness due to the confusion and misuse caused by such confusion. A PDFD model is concerned with those aspects of a system that influence how the processes, data stores, and data flows are implemented (PDFD data flows represent actual processes and the movement of data). A PDFD is therefore an implementation *dependent* view created for the analysis of a system. A LDFD model, on the other hand, is an implementation *independent* view created for the design of a system. LDFDs show only the essential features of system being modeled. Implementation dependent processes found in the PDFD are omitted in the LDFD view. In fact, implementation details are explicitly avoided.

7.5 Tips and Traps

The following observations have been made about data flow diagrams [DeMarco 79]:

1. *How do DFDs differ from system flowcharts?* DFDs show the flow of data while system flowcharts show the flow of control. DFDs also present the design philosophy progressing from the abstract at the upper levels to the concrete at the lower levels and hence, unlike flowcharts, are used as specification tools.

2. *How many levels should be expected?* Although dependent on system size and the extent of partitioning at each level, ten levels would probably be a good cutoff for a leveled DFD. With ten levels, one should be able to model quite large systems.

3. *When looking at the details of level n, is modification of the n-1 level often required?* Yes. Usually, however, the resulting ripple effect only goes up one level.

4. *What if it is difficult or impossible to get started on a pure top-down analysis?* If it is too difficult to see the big picture, the middle might be a good place to start. After collecting all of the middle-level pictures, combine them into one diagram and then try building the

top level.

5. *How can the flow of physical goods be represented on a DFD if one is restricted to pure data?* Although it is often difficult to separate the two, often objects have data content. This could be part numbers, a count of the objects, or some other form of data content. In a hospital, for example, patients would have their ages, blood types, and pulse rates as forms of data content. The doctors and nurses would still not show up on the DFD since they are processing the data.

Other common errors include processes that have inputs but no outputs and processes that have output but no input. All processes must have at least one data input and one data output.

7.6 Integration With Other Methodologies

The idea behind integrating methodologies is to use the information contained in one methodology, such as DFD, to build a model in another methodology such as IDEF0. The advantage of this lies in the different views of a system that each methodology provides.

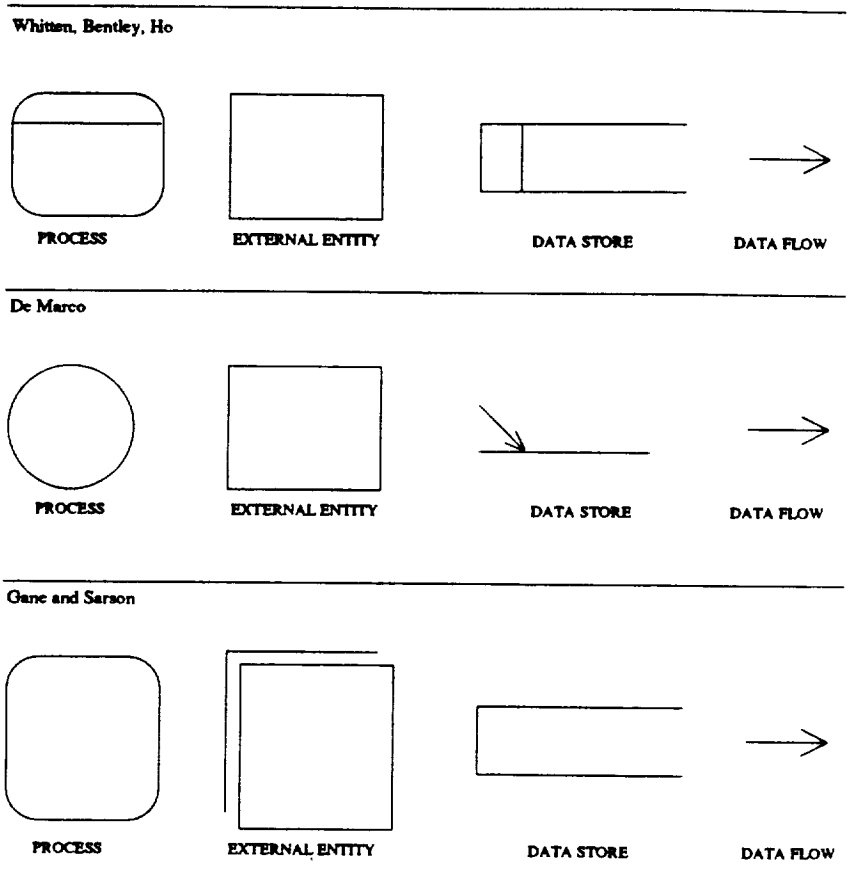
Integration with other methodologies will initially address IDEF0 to LDFD, LDFD to IDEF0, and PDFD to IDEF0 because IDEF0 has similar semantics to DFD. The integration of IDEF0 to PDFD will not be addressed initially since it requires automation of the design process which we anticipate to be the most difficult.

7.7 Conclusions

DFD provides the modeler with a methodology for modeling a system from the viewpoint of the data. It was designed to avoid the bias in models normally created when a person or department's viewpoint is taken and to capture a global view of the data.

Integration with other methodologies will focus initially on integration with IDEF0 and Structure Charts as the pursuit for more general integration strategies continues.

Appendix A. Notational Conventions



DFD Notational Conventions

Data Flow Diagrams are also referred to as DFD's, Data Flow Graphs, and Bubble Charts.

Appendix B. Abbreviations used in the DFD Metamodel

CONTEXT: context

DATA-STORE: data store, file

DECOMPOSITION: decomposition

DF-BTWN-P&EE: connects process and external entity

DF-BTWN-P&DS: connects process and data store

DF-BTWN-P&P: connects process to process

DIRECTION: indicates the source to destination direction of the DF

DSN: data store name

DS-OCC-IN-DEC: data store occurrence in decomposition

EXTERNAL-ENT: external entity

EE-OCC-IN-DEC: external entity occurrence in decomposition

EEN: external entity name

FROM: source process for process to process connection

LID: link id

LINK: link

LINK-OCC-IN-DF1: link occurrence association with a data flow between a process and a data store

LINK-OCC-IN-DF2: link occurrence association with a data flow between a process and an external entity

LINK-OCC-IN-DF3: link occurrence association with a data flow between two processes

NAME: name

OCC: occurrence number

PACK-IN-STRUCT: packet in structure

PID: process id

PROCESS: process

PRO-OCC-IN-DEC: process occurrence in decomposition

ROLE: role

SID: structure id

STRUCTURE: for splits and joins

TO: destination process for process to process connection

References

DeMarco, Tom. "Structured Analysis and System Specification." New York: Prentice-Hall, 1979.

Gane, Chris and Sarson, Trish. "Structured Systems Analysis: tools and techniques," New York: Improved Systems Technologies Databooks, 1977.

Ross, Douglas T., "Structured Analysis (SA): A Language for Communicating Ideas." *IEEE Transactions on Software Engineering*, January 1977.

SofTech, "Integrated Computer-Aided Manufacturing (ICAM) Function Modeling Manual ($IDEF_0$)," Technical Report UM110231100, June 1981.

Whitehouse, G.E. "Systems Analysis and Design using Network Techniques," Prentice-Hall, 1973.

Whitten, Bentley, and Ho. System Analysis and Design Methods. St. Louis: Times Mirrow/Mosby College Publishing, 1986.

Yourdon, E. and Constantine, L.L., "Structured Design," Yourdon Inc., 1975.

8

Structure Charts: Modeling the Referential Structure[1]

Since the mid '70s, programmers and analysts have realized the importance of designing programs *before* coding takes place. Just as an architect completely specifies the plans of a building and builds models before construction begins, the designer of large computer programs must develop design specifications and create models before coding starts. As computer programs grow increasingly complex, poorly designed programs become unmanageable with higher maintenance and modification costs and unreliable performance . Structure charts were developed to graphically document the hierarchical relationships between modules in computer programs. Moreover, structure charts were designed to promote modularity and data hiding and to highlight poorly designed referential structures.

This chapter will briefly orient the reader to the purpose, syntax, and semantics of structure charts. Next, the Integrated Computer-Aided Manufacturing (ICAM) DEFinition (IDEF) language $IDEF_1$ information model of the structure chart methodology will be presented. This will be followed by an examination of the strengths and weaknesses of the methodology. Finally, some hints for using structure charts and an evaluation of integration strategies with other methodologies will be presented.

[1] Funded in part by Tandem Computer Incorporated

8.1 What are Structure Charts?

Understanding structure charts can be approached by clarifying the differences between structure charts, flowcharts, and data flow diagrams (DFDs). Although the motivations for each are similar, the methodologies are quite different.

Flowcharts model the flow of control while only implicitly representing the referential structure. They do not model the flow of data. DFDs, on the other hand, model the flow of data rather than the flow of control. A DFD is a declaration of the data flow requirements of a system [DeMarco 79]. While flowcharts may implicitly represent the referential structure, DFDs do not.

Structure charts were developed to aid in the design of structured programs by graphically representing the hierarchical relationships between the modules that compose a program. Therefore, by definition, the structure chart methodology focuses on the referential structure of the program's modules. Structure charts do not directly model the flow of control or the flow of data. Furthermore, structure charts reveal nothing about the decision structure or the order in which subordinate modules are called (except for cases of parallel activation or coroutines). Basically, structure charts represent how a system is partitioned into modules and the interfaces between those modules.

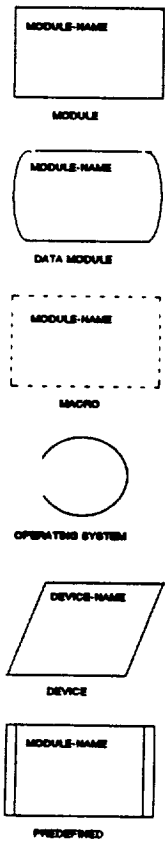


Figure 8.1 Types of Modules

8.2 Syntax and Semantics

With a general idea of the purpose of structure charts, a brief look at the structure chart syntax and semantics should provide insight into the methodology. Appendices A and B of Constantine and Yourdon's book provide an excellent reference for the structure chart methodology [Constantine 79]. Constantine and Yourdon's convention for defining and drawing structure charts will be used throughout this discussion. Structure charts were designed to represent the structural features of computer programs regardless of language or environment. Modules, connections, and couples are the primary graphic elements used by structure charts.

8.2.1 Modules

Modules are the building blocks of modular computer systems and the primary element of structure charts (see Figure 8.1). The notational representation for generic modules is a simple rectangle. The module name is placed in the upper-left corner of the module. The module name represents the lexically contiguous statements and any lexically included statements included in the module [Constantine

79]. In other words, the module name represents a block of code and any subprograms it references as a single entity.

Different types of modules may be represented by variations of the simple rectangle notation. For example, dashed line rectangles represent modules defining macros. The five basic types of modules are 1) normal, 2) data, 3) macro, 4) operating environment, and 5) device [Constantine 79]. A module denoted by a plain rectangular box may represent any module regardless of physical or activation characteristics. Vertical stripes at each end denote predefined modules. A module whose contents consist exclusively of data is denoted by a plain rectangular box whose vertical ends are bowed in an outward direction. A macro is represented by a plain rectangular dashed box. Macros are modules that are inserted or expanded in-line at compile-time. Devices consist of input-output mechanisms (disk drives, card readers, printers, etc.) and files among other things. They are represented by parallelograms that lean toward the right. Finally, the operating environment for the system is denoted by a circle with a small section removed from the left side symmetically along the horizontal axis. The operating system, hardware, and system management are included within the operation environment.

8.2.2 Intermodular Connections and Couples

An intermodular connection is a reference by one module to another module or to the identifier of some element within the other module (see Figure 82). If a connection is a recursive call the module could actually refer to itself. The types of connections are 1) subordination, 2) cotransfer, 3) subordinated cotransfer, 4) transfer, 5) data transfer, 6) data reference, 7) control reference, and 8) hybrid reference. There are also asynchronous versions of subordination, cotransfer, subordinated cotransfer, transfer, and control reference [Constantine 79].

Subordination is either a subroutine call, function reference, or macro invocation. Cotransfer consists of one module referencing another as a coroutine. Transfer is merely an unconditioned direct transfer of control from one module to another by name. Data transfer differs from transfer in that only data is transferred and not control from one module to another. Data references are pathological connections in which one modules references an identifier in another module. Control references and hybrid references are also pathological in nature. A control reference is essentially a "goto". Specifically, a reference is made in one module to an identifier in a second module as a means of transfering control. In a hybrid reference, one module affects the procedure of another module.

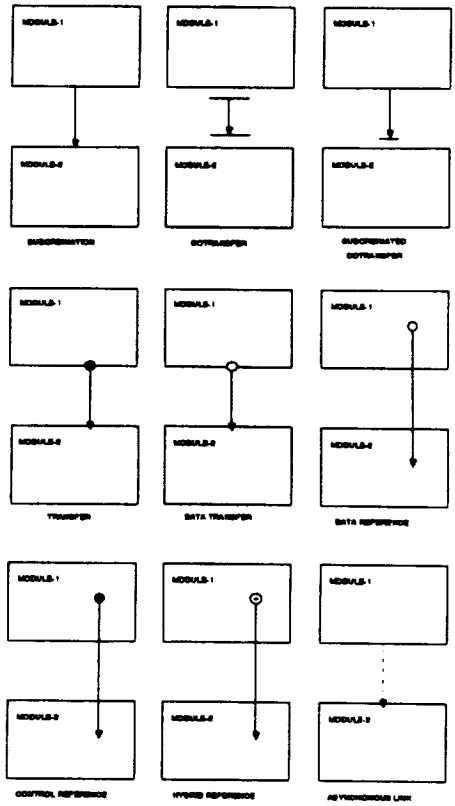


Figure 8.2 Intermodular Connections

Notationally, lines represent intermodular connections (see Figure 8.3). Normal connections are represented by lines that begin and end at the edges of the modules (rectangles). Annotating a line with an arrowhead indicates the direction of the flow of data. Names of parameters being passed may be added near the connection line. Adding an open circle to the tail of an annotated arrow indicates the parameters are data. Alternatively, adding a filled circle to the tail indicates the parameters are for control purposes. Passing the name of an employee to a module would be an example of data while passing an error flag would be an example of a control parameter.

Coupling is a measure of the interdependence of modules [DeMarco 79]. A couple is a piece of information that flows between modules via connections. Couples are represented by short arrows with open or closed circular tails drawn parallel to the intermodular connections. A direction is therefore associated with a couple representing the direction that the information is flowing. The information will be either control information or data. An intermodular connection may have multiple couples associated with it.

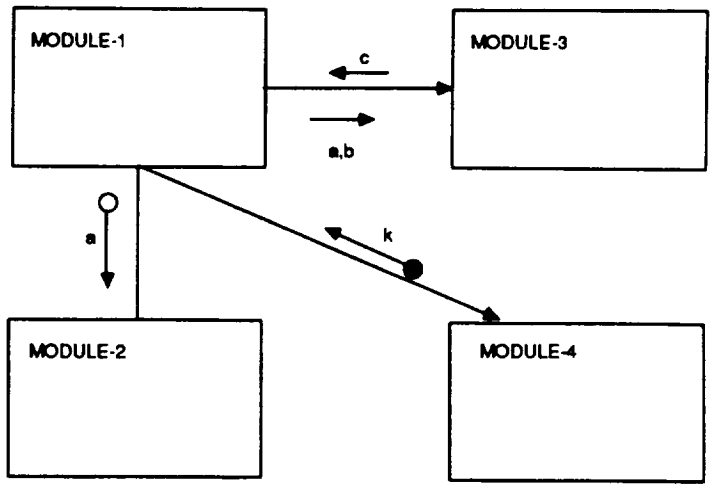


Figure 8.3 Annotations for Connections

Cohesion and coupling are interrelated concepts. In general, as cohesion increases within individual modules, the coupling between those modules decreases. Cohesion has been called the cement that holds the processing elements of a module together [Constantine 79]. It is a measure of the functional relatedness between elements that are in a module. The range of possible levels of cohesion, from least to most desirable, are clasified as coincidental, logical, temporal, procedural, communicational, sequential and functional cohesion. Cohesion levels provide a powerful tool that gives the designer a feel for the degree that elements in a module are bound or related, and thus the degree of coupling of inherent in the proposed design. The major drawback of this approach to design analysis is its subjective nature.

Connections, other than the types already given, may also be divided into two general categories: normal and pathological. Normal connections occur when a reference is made to the module name. Pathological connections are, by definition, potentially unhealthy connections. A pathological connection can be an intermodular connection in which a direct reference by one module is made to data of another module or a direct transfer of control is made between modules. A module that relies on the value of a variable calculated in another module is one example of a pathological connection. Notationally, pathological connections are represented by lines with annotated arrows as before except the lines begin and end within the rectangles representing the modules. Pathological connections are

one example where the graphical notation of structure charts can highlight potential problems.

Lexical relationships, such as inclusion and contiguity, may also be expressed in structure charts. Lexical inclusion represents the fact that one module may be completely within the lexical boundaries of another module. Lexical contiguity represents the fact that two modules have an order associated with them. Since lexical relationships are fixed at definition time, strong interdependencies may be introduced between modules by the lexical structure of the program [Constantine 79]. Lexical interrelationships are often referred to as content-coupling and represented by the overlapping of modules.

8.2.3 Procedural Annotations

Control structures may also be associated with modules to convey procedural information about the connections that the module makes. The three types of control structures are loop, decision, and goto.

To represent intermodular references from within a loop construct, a line is drawn from inside one end of the calling module through the intermodular connections enclosed inside the loop and terminating inside the calling module. An arrowhead at one end of each connection (line) distinguishes the direction of the calls. The arrowhead end of the connection points to the referenced module. The annotation for references within a loop can be nested in any combination that loops themselves can be nested.

Intermodular references which depend on the result of a condition or decision to occur may be represented by enclosing the originating end of the connection (line) in a diamond. More than one intermodular reference may originate from the diamond if multiple intermodular references occur due to a single condition or decision.

The last control structure is the goto. It simply represents a one way connection in which no return to the module is planned for in the structure.

8.3 Metamodel

The metamodel of the structure chart methodology was developed using $IDEF_1$. In this chapter, the structure chart methodology is being treated as a system and the information managed by this methodology is being modeled using $IDEF_1$ (see Figure 84).

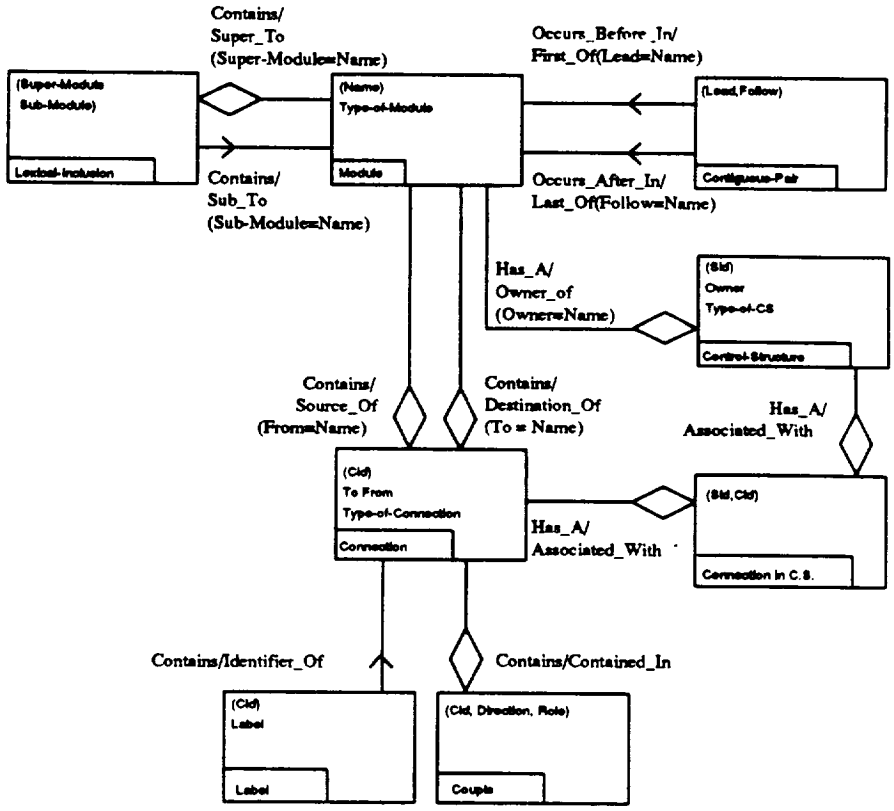


Figure 8.4 IDEF1 Structure Chart Metamodel

8.3.1 Modules

As discussed earlier, modules are the primary element of structure charts. The module entity class (see Figure 8.5) therefore provides a logical starting point for the structure chart metamodel. Modules maintain certain information, specifically, an identifying name and the type of module (i.e. normal, data, macro, operating environment, device, predefined normal, predefined macro, or predefined data). The module name is unique and therefore sufficient to form the key class for the module entity class.

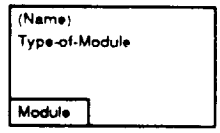


Figure 8.5 Metamodel of modules

8.3.2 Lexical Relationships

Lexical inclusion between modules reveals structural information about module pairs. A lexically inclusive relationship between two modules involves one module being completely included within the other. This is a super-module and sub-module relationship. The important information to manage is the name of the super-module and the name of the sub-module. The *lexical-inclusion* entity class (see Figure 8.6) may now be added to the metamodel. Its key class will consist of the names of the super-module and sub-module.

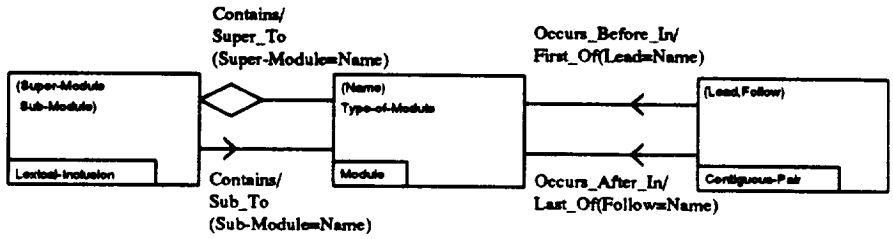


Figure 8.6 Metamodel of lexical inclusion

Lexical adjacency of modules also reveals structural information about module pairs. A lexically adjacent relationship between two modules involves their ordering. As with lexical inclusion, the names of the two modules is the only pertinent information to maintain. In this case, however, the name of the "lead" module and the name of the "follow" module are used to distinguish the order. The *contiguous-pair* entity class may now be added to the metamodel with its key class consisting of lead and follow.

8.3.3 Intermodular Connections

Although modules are the primary building blocks for structure charts, the intermodular connections are just as important. Recall that an intermodular connection represents a reference or call to a module from inside another module. This can occur with or without arguments being passed. Besides the name of the calling module and the name of the called module, the type of connection must be maintained. Recall that the types of connections are subordination, cotransfer, subordinated cotransfer, transfer, data transfer, data reference, control reference, and hybrid reference. In addition the asynchronous versions of subordination, cotransfer, subordinated cotransfer, transfer, and control reference are also possible. The called module name will be referred to as the *To* module and the calling module name will be referred to as the *From* module. The *Connection* entity class (see Figure 8.7) may now be added to the

structure chart metamodel. Its attribute classes will consist of the *To* module, the *From* module, and the type of connection. A key class is now needed to uniquely identify the entity class. Note that none of the items in the attribute class were sufficient either singly or in combination to uniquely identify the connection entity class. A connection id or Cid will be created to serve as the key class.

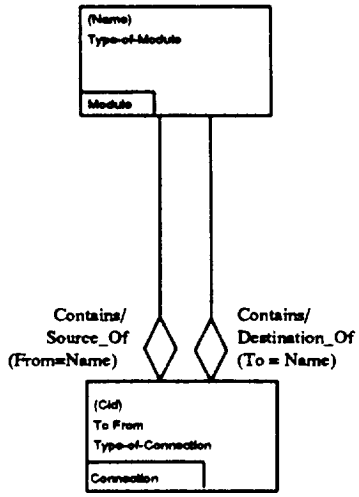


Figure 8.7 Metamodel of connection

8.3.4 Couples

Connections are often annotated with one or more couples. They represent the fact that data items are part of the connection. Recall that couples provide a measure of the interdependence of modules and consist of short arrows. Additionally, couples with open or filled circular tails represent information about the role of the arguments. The role may be control either information or data. A couple also has one or more parameters and a direction associated with it. The *Couple* entity class (see Figure 8.8) may now be added to the structure chart

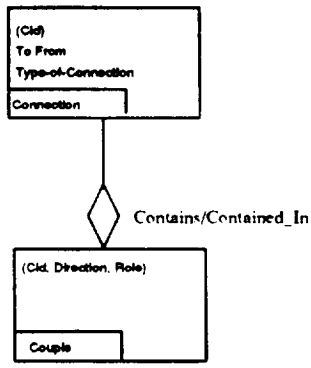


Figure 8.8 Metamodel of couple

metamodel with a zero, one or many relationship to the *connection* entity class. By inheriting the key class from the *connection* entity class Cid and combining it with direction, and role, a uniquely identifying key class is created for the *Couple* entity class.

8.3.5 Labels

A connection may or may not have a label. A label is an entry point to a module used by control references. A label does not have to be unique. This information is kept about each connection but cannot be included as an the attribute class of the *Connection* entity class due to the no-null rule in IDEF$_1$. A *label* entity class (see Figure 8.9) will therefore be created and added to the structure chart metamodel with a zero or one relationship to the *Connection* entity class. The key class of the *Connection* entity class Cid will be inherited and label will be the only additional attribute class.

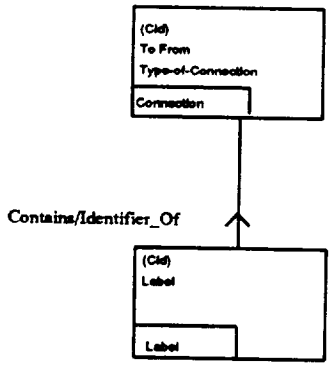


Figure 8.9 Metamodel of label

8.3.6 Control Structures

There is one facet of modules that has not been addressed. A module may have zero, one or many control structures associated with it. A control structure is an annotation to a module that conveys procedural information about the connections that the module makes. The name of the module containing the control structure is the first piece of information that must be kept. It will be referred to as the Owner. The type of control structure (type-of-CS) will also need to be maintained. Decision, loop, and goto are the three possible types of control structures. Since each module may have zero, one, or many control structures, the information cannot be kept by the *Module* entity class (see Figure 8.10). This would violate the no-repeat and no-null rule. A *Control-Structure* entity class will therefore be added to the structure chart metamodel. Neither the Owner nor Type-of-CS are sufficient to form the key class. The control-structure key class will

therefore consist solely of a unique name (Sid) assigned upon creation of the entity class. Owner and Type-of-CS will serve as non-key classes.

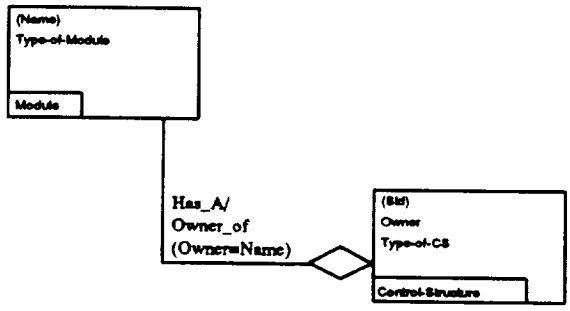


Figure 8.10 Metamodel of control structure

8.3.7 Connections in Control Structures

The discussion of the structure chart metamodel is now almost complete. The last detail to consider is that a connection may be in zero, one or many control structures and that control structure may contain one or many connections. The *Connection-in-CS* entity class (see Figure 8.11) will now be added to the structure chart metamodel. It will establish zero, one, or many relationships with the *Control-Structure* entity class and the *Connection* entity class. The key class will consist of Sid inherited from the *Control-Structure* entity class and Cid inherited from the *Connection* entity class.

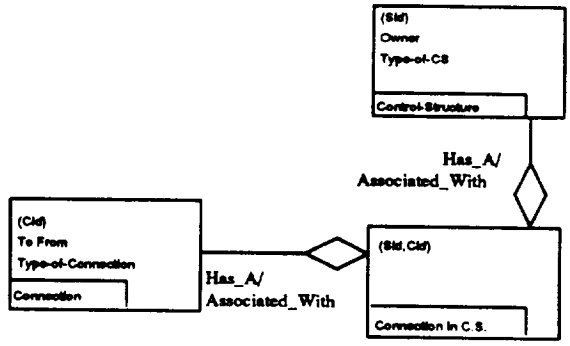


Figure 8.11 Metamodel of connections in control structures

8.4 Strengths and Weaknesses

Structure charts may also be effectively used with other methodologies. Using structure charts with Entity Relationship diagrams and DFD forms a powerful design methodology for many business application systems.

Coupling and cohesion are two design characteristics which structure charts graphically highlight (coupling is concerned with dependence between modules while cohesion is concerned with intramodule association).

Coupling provides a measure of independence between modules and has a strong effect on the readability of both the design and code of modules [Myers 75]. The higher the coupling between modules, the greater the likelihood of side effects when making modifications to those modules.

Cohesion pertains to the strength of association of the elements inside a module. A module with high cohesion contains elements that are closely related and can be naturally treated together. A module with low cohesion, on the other hand, contains elements with unrelated elements. High cohesion is desired for modular, readable designs and code.

Normal and pathological connections are also graphically highlighted by structure charts. Recall that pathological connections are potentially unhealthy connections that indicate a design with greater likelihood for structural problems.

A major weakness of structure charts is its inability to model object oriented programming applications.[2] Features such as multiple or single inheritance and polymorphism will require additional graphical representations. Structure charts also cannot model program data structures. The types of problem areas in design that structure charts are designed to graphically highlight is not sufficient in the world of object-oriented programming.

8.5 Tips and Traps

The following observations have been made about structure charts:

A structure chart with crowds of couples is a sure sign of poor partitioning. Overcoupling generally indicates poor cohesion[DeMarco 79].

The four major factors that tend to influence coupling are 1) the types of connections between modules, 2) the complexity of the interface, 3) the type of information flow along the connection, and 4) the binding time of the connection [Constantine 79]. The complexity of the interface refers to the number of arguments passed between

[2] The object-oriented design method IDEF4 was developed to model object-oriented systems.

modules. The types of information that flow along a connection are data, control, and a combination of the two. Constantine claims that control communication is dispensable and that the communication of data alone is necessary for functioning systems of modules. Coupling is minimized when only data flows between interfaces. Modules using control communication must be less independent and therefore more highly coupled. The binding time of the connection refers to connections bound at execution, linking, compilation, or during coding. The earlier (and therefore longer) a connection is bound, the higher the coupling tends to be between the calling and called modules.

Control information is often disguised as data information in the form of flags or an address. Coupling is increased whenever control information is passed between modules.

Global data greatly increases intermodular coupling. By making data only accessible to those modules that need access, coupling can be greatly decreased.

Modules coupled with a single input couple and/or a single output couple will generally have the strongest cohesion. Poor cohesion is often indicated by modules with downward passing switches for control purposes and by modules that require pathological data communications[DeMarco 79].

8.6 Integration With Other Methodologies

The following observations have been made about the correlation between DFDs and structure charts [DeMarco 79]. A DFD is a statement of requirement declaring *what* has to be accomplished while a structure chart is a statement of design declaring *how* the requirement shall be met. The relationship between the two reflects the relationship between intent and method.

Transform analysis and transaction analysis are two methods from structured design for deriving a structure chart from a DFD. This is important because these methods embody a design process that can be automated. Insights from these methods combined with the results from our structure chart metamodel should provide a solid platform for developing an integration strategy between structure charts and data flow diagrams. The question of how to derive a data flow diagram from a structure chart could be solved by collecting the additional information required by the data flow diagram when constructing the structure chart.

8.7 Conclusions

The information that is actually maintained by the structure chart methodology is considerably less than that maintained by the data flow diagram methodology. This is not surprising since manual methods exist for deriving structure charts from data flow diagrams.

Data flow diagrams are also closely related to IDEF$_0$ models in the information they keep. At this point, efforts will be concentrated on isolating the common pieces of information maintained by structure charts, data flow diagrams, and IDEF$_0$ and developing an integration strategy among these three methodologies.

Appendix A. Abbreviations used in the Structure Chart Metamodel

Cid: connection id; a unique identification number assigned to each connection upon its creation in a structure chart

Connection: entity class that maintains the name of the calling module, the name of the called module, and the type of connection between the two modules

Connection-in-CS: establishes a zero, one or many relationship with the control-structure entity class and the connection entity class

Contiguous-Pair: entity class needed to maintain the names of two modules with a lexically adjacent relationship. Lead and Follow are used to distinguish the order

Control-Structure: entity class needed to maintain the name of the module containing the control structure (owner) and the type of control structure (type-of-CS)

Follow: inherited name of the module that follows the other in contiguous-pair. See contiguous-pair entity class

From: inherited name of referenced module

Label: (entity class) an entry point to a module used by control references in structure charts

Label: (attribute) identifier of the entry point

Lead: inherited name of the module that precedes the other in the contiguous pair

Lexical-Inclusion: entity class

Module: entity class

Name: name by which a module is referrenced

Owner: name of the module in which the control-structure occurs

Role: the type of information being passed; either control information or data

Sid: a unique tag assigned to each control-structure upon its creation

Sub-Module: included in the key class of the Lexical-Inclusion entity class; inherited name of the included module

Super-Module: included in the key class of the Lexical-Inclusion entity class; inherited name of the including module

To: inherited name of the referencing module

References

Constantine, L.L. and Yourdon, E. Structured Design. Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

DeMarco, Tom. Structured Analysis and System Specification. New York: Prentice-Hall, 1979.

Myers, Glenford J. Reliable Software Through Composite Design. New York: Petrocelli/Charter, 1975.

Glossary of Important Terms

The following terms appear in this report:

Activity ($IDEF_0$): a function with inputs, controls, outputs and mechanisms.

Alternate Key (IDEF1x): a collection of attributes which uniquely identify an instance of an entity. Alternate keys cannot be inherited across relations.

Attribute: a data element which maintains information about, characterizes or describes an entity (IDEF1x). In ER attributes are functions which map an entity set to a Cartesian product of value sets.

Attribute Class ($IDEF_1$): maintains information about, characterizes or describes characteristics of an entity class.

Bridge Type (ENALIM): an association between a LOT and a NOLOT.

Candidate Key (ER): an attribute or group of attributes whose values uniquely identify every tuple in a relation. No attributes can be removed from the candidate key without destroying the unique identification.

Cardinality: the number of instances of the dependent entity (class, type or set) that are related to a single instance of an independent entity.

Categorization Relation ($IDEF_{1x}$): defines generalization-specialization relationships.

Cohesion: a measure of the inter-modular dependence.

Concept ($IDEF_0$): something that is produced and/or consumed by an activity. Concepts can be either actual things or abstract ideas, or information objects.

Conceptual Schema: a logical or non-specific representation of the data to help provide a basis for the mapping of the external and internal schemas.

Connection-relationship ($IDEF_{1x}$): a link relating two entities originating at the parent or independent entity and terminating at the child or dependent entity. (similar to a link class in $IDEF_1$).

Constraint: a predicate logic statement about objects and relations in a model which must hold in order for a model to be valid.

Converging Data Flow (DFD): many sources and a single destination. Similar to many-to-one relationships.

Couple: information flowing between modules via links.

Coupling: a measure of modular interdependence.

Data Flow Diagram: a network view of the system showing the flow of data, the transformation of data by processes, the storage of data by data stores and the sources and/or destinations of data from outside the context of the system referred to as external entities.

Decomposition ($IDEF_0$): view of three to six sub-functions of an activity and their relationships.

Dependent Entity Class ($IDEF_1$): an entity class whose existence depends on the existence of the independent entity class.

Discriminator Attribute ($IDEF_{1x}$): a characteristic which distinguishes categorized entities in a generalization-specialization relationship.

Diverging Data Flow (DFD): one source and multiple destinations. Similar to a one-to-many relationship.

Data Store (DFD): a temporary storage place for data.

Entity ($IDEF_{1x}$): a type of real or abstract object.

Entity Class ($IDEF_1$): a class of real or abstract objects about which information is kept.

Entity Key (ER): a group of attributes such that the mapping from the entity set to the corresponding group of value sets is one-to-one.

Entity Set (ER): a group of like or similar entities. Entity sets may or may not be mutually disjoint.

External Entity (DFD): a source or destination of data outside the context of the system.

External Schema: an abstract user or application view of an environment or system.

Existence Dependence (ER): a child entity must have a parent entity for it to exist.

Fact Type (ENALIM): a link or association between two ENALIM objects.

Foreign Key ($IDEF_{1x}$): primary key or keys inherited from other entities.

Idea Type (ENALIM): it is an association between two NOLOTs.

Identification Dependence (ER): identification of the child entity depends on the identification of the parent entity on which the child's existence is dependent.

Identifier Constraint (ENALIM): a constraint placed on an object role pair which states that the role pair uniquely identifies an instance of the fact type or association.

Identifier-dependent ($IDEF_{1x}$): an entity which depends on other entities in order to have meaning.

Identifier-independent ($IDEF_{1x}$): an entity which can exist independent of other entities.

Independent Entity Class ($IDEF_1$): an entity class which can exist independent of other entity classes.

Inheritance ($IDEF_1$): when two entity classes are linked via a link class relationship, the attributes occuring in the key class of the independent entity class are inherited by the dependent entity class.

Inherited Attribute Class ($IDEF_1$): attributes inherited across a link class, from an independent to a dependent entity class.

Intermodular Connections (Structure Charts): indicates a reference or call made by one module to another.

Internal Schema: refers to the actual implementation of the data or the database schema.

Key Class ($IDEF_1$): a collection of attribute classes which uniquely identify an instance of an entity class.

Label (Structure Charts) : the point of entry into a module.

Leveling (DFD): pertains to the breaking down of a process into finer detail. Similar to decomposition in $IDEF_0$ diagrams.

Lexical Contiguity (Structure Charts): when two modules may have an order associated with them.

Lexical Inclusion (Structure Charts): when one module is completely within the lexical boundaries of another.

Link Class ($IDEF_1$): a binary relationship between two entity classes.

LOT (ENALIM): Lexical Object Type, denotes concepts or objects which can be processed by an information system.

Metamodel: a model of the information kept by a methodology.

Modular Intradependence: relationship of elements within a module.

Modular Interdependence: relationship of elements in different modules.

Modules: building blocks of modular computer systems and the primary entity of structure charts.

NOLOT (ENALIM): Non-Lexical Object Type, denotes concepts or objects which cannot be directly processed by the system.

Non-specific Relations ($IDEF_{1x}$): a many-to-many or one-to-one relationship.

Object Types (ENALIM): a collection of objects which are grouped together in a class hierarchy.

One to Zero or One Link ($IDEF_1$): an instance of an independent entity class can be associated with either zero or one instances of the dependent entity class.

Owned Attribute Class ($IDEF_1$): attributes which are owned by an entity class, (they originate from that entity class).

Path ($IDEF_0$): a link between two activities, which relate a producer activity to a consumer activity.

Primary Key ($IDEF_{1x}$): a collection of attributes which uniquely identify an instance of an entity. Primary keys are inherited across relations from independent to dependent entity.

Process (DFD): transforms data flows showing some type of work performed on the data.

Regular Entity Relation (DFD): relation where entities are completely identified by their entity key.

Regular Relationship Relation (DFD): relation where all entities are identified by their owned attributes.

Relation (ER): a table of related objects.

Relationship Relation (ER): an organization of information of related entities.

Relationship Set (ER): similar to an entity set, it refers to a classification of like or similar entities.

Role ($IDEF_{1x}$): the function an entity performs in a relationship. A role describes the use of an attribute in an entity. The role name is descriptive of the role.

Role Equality Constraint (ENALIM): a constraint which specifies that the set of instances of two roles must be equivalent.

Role Exclusion Constraint (ENALIM): a constraint which says that the set of instances of two roles must be mutually exclusive.

Role Name ($IDEF_{1X}$): the name of the relation which is placed in front of an inherited attribute to distinguish between similar attributes inherited via different relations.

Role Subset Constraint (ENALIM): a constraint which states that the set of instances of one role must be a subset of the set of instances of another role.

Role Uniqueness Constraint (ENALIM): a constraint which states that an instance of an object is uniquely identified by the combination of two or more role pairs.

Strong One to Many Link ($IDEF_1$): an instance of an independent entity class must be associated with at least one instance of the dependent entity class.

Structure ($IDEF_0$): a metamodel artifact which models spreads, splits, joins and bundles of concepts.

Structure Charts: focuses on how a program is to be partitioned into modules and on the interfaces between the different modules.

Subtype Exclusion Constraint (ENALIM): a constraint which states that the sets of instances of two subtypes of an object are mutually exclusive.

Subtype Total Constraint (ENALIM): a constraint which declares that the union of the sets of instances of all the subtypes of an object, constitute the set of instances of the supertype.

Total Role Constraint (ENALIM): a constraint which states that there must be at least one instance for every object type playing a role.

Tunneled Concept ($IDEF_0$): a concept which enters the model from the environment or leaves the model to the environment without propagating through the parent level activities.

Value Set (ER): a classification of similar values.

Weak Relationship Relation (ER): a relation upon which entities depend in order to be completely identified.

Weak One to Many Link ($IDEF_1$): an instance of an independent entity class can be associated with either zero, one or many instances of the dependent entity class.

Index

Copies of this publication have been deposited with the Texas State Library in compliance with the State Depository Law.