CHAPTER 10

The IDEF Family of Languages

Christopher Menzel, Richard J. Mayer

The purpose of this contribution is to serve as a clear introduction to the modeling languages of the three most widely used IDEF methods: IDEF0, IDEF1X, and IDEF3. Each language is presented in turn, beginning with a discussion of the underlying "ontology" the language purports to describe, followed by presentations of the syntax of the language — particularly the notion of a model for the language — and the semantical rules that determine how models are to be interpreted. The level of detail should be sufficient to enable the reader both to understand the intended areas of application of the languages and to read and construct simple models of each of the three types.

1 Introduction

A modeling method comprises a specialized modeling *language* for representing a certain class of information, and a modeling *methodology* for collecting, maintaining, and using the information so represented. The focus of this paper will be on the languages of the three most widely used IDEF methods: The IDEF0 business function modeling method, the IDEF1X data modeling method, and the IDEF3 process modeling method.

Any usable modeling language has both a syntax and a semantics: a set of rules (often implicit) that determines the legitimate syntactic constructs of the language, and a set of rules (often implicit) the determines the meanings of those constructs. It is not the purpose of this paper is to serve as an exhaustive reference manual for the three IDEF languages at issue. Nor will it discuss the methodologies that underlie the applications of the languages. There are other sources that discuss these issues ([NIST93a, NIST93b, MMP93]). Rather, the purpose of this paper is simply to serve as a clear introduction to the IDEF languages proper, that is, to their basic syntax and semantics. It is thus hoped that the paper will quickly enable the reader both to understand the intended areas of application of the languages and, more specifically, to read and construct simple models of each of the three types.

2 Background to the IDEF Languages

The IDEF suite of modeling languages arose in the 1970s out of the U.S. Air Force Integrated Computer Aided Manufacturing (ICAM) program. The goal of ICAM was to leverage computer technology to increase manufacturing productivity. A fundamental assumption of the program was the need for powerful but usable modeling methods to support system design and analysis. Consequently, the program undertook the development of a suite of "ICAM DEFinition," or IDEF, methods. These included an activity, or "function," modeling method (IDEF0), a conceptual modeling method (IDEF1), and a simulation model specification method (IDEF2). IDEF0 was based loosely upon the Structured Analysis and Design Technique (SADT) pioneered by Douglas Ross [Ros77] and IDEF1 upon the Entity, Link, Key Attribute (ELKA) method developed chiefly at Hughes Aircraft by Timothy Ramey and Robert Brown [RB87]. Since the ICAM program there have been several important developments. First, in 1983, the Air Force Integrated Information Support System (I^2S^2) program added several constructs to the IDEF1 method that were felt to make it more suitable as a database schema modeling method. The result was IDEF1X, which is now more widely used than IDEF1. Beginning in the late 1980s, work began on a process modeling method known as IDEF3, and was completed under the Air Force Information Integration for Concurrent Engineering (IICE) program. IDEF3 subsumes much of the original role of IDEF2, as it can be used for the specification of effective first-cut simulation models. Additionally, the IDEF3 language has an object-state component that can be used for modeling how objects undergo change in a process. The early 1990s saw the emergence of IDEF4 and IDEF5. IDEF4 is an object-oriented software design method that integrates requirements specified in other methods through a process of iterative refinement. It also supports the capture and management of design rationale. IDEF5 is a knowledge acquisition and engineering method designed to support the construction of enterprise *ontologies* [Gru93]. Because of space limitations, these newer methods will not be discussed further in this paper. Interested readers are referred to [MKB95] and [MBM94].

Recent developments have focused on refinement and *integration* of the IDEF languages. That is, the focus has been on the development of both theory and techniques to support the easy exchange of information between different IDEF (and non-IDEF) models, and, ultimately, on the automated exchange and propagation of information between IDEF (and non-IDEF) modeling software applications. To reflect these developments, "IDEF" is now usually taken to be an acronym for *Integration DEFinition.*

The IDEF0, IDEF1X, and, increasingly, IDEF3 methods are widely used in both government and the commercial business sectors. The focus of this paper will be on the languages of these methods. In many presentations of one or another IDEF language, syntax and semantics are intermingled so as to make them difficult to distinguish. A goal of this paper is to keep this

distinction sharp. Thus, each major section begins with a discussion of the basic semantic, or *ontological*, categories of the method at hand, independent of any syntactic considerations. Only then is the syntax of the language introduced, first its *lexicon* (i.e., its more primitive elements), then its *grammar* (i.e., the rules determine how complex expressions are ultimately built up from the elements of the lexicon).

3 The IDEF0 Function Modeling Language

We begin with the IDEF0 function modeling language, the method for building models of enterprise activities.

3.1 The IDEF0 Ontology: Functions and ICOMs

In general, an activity is a *thing that happens*, whether (in effect) instantaneously or over some (possibly fragmented, discontinuous) period of time. Simple examples of activities include the death of Caesar, Jessie Owens' running of the 100 yard dash in the finals of the 1936 Olympics, and the writing of this paper. In IDEF0 modeling, however, attention is often focused not just on actual "as-is" activities, but *possible* activities as well — the activities of a merely envisioned company, for example, or those of a proposed virtual enterprise. Thus, one might say, the primary focus of IDEF0's ontology — the things that exist according to IDEF0 — is the class of all possible activities, whether actual or not. However, it is not concerned with just any sort of activity, but with a certain kind, known in IDEF0 as a *function*. Thus, IDEF0 is often referred to as a "function modeling method." An IDEF0 function is a special kind of activity, namely, one that, typically, takes certain inputs and, by means of some mechanism, and subject to certain controls, transforms the inputs into outputs — note the parallel with the notion of a mathematical function wherein a given set of *arguments* (inputs) is "transformed" into a unique *value* (output). (That noted, we shall follow common practice and usually use the generic term 'activity'.) The notions of input and output should be intuitively clear. Controls are things like laws, policies, standards, unchangeable facts of the environment, and the like that can guide or constrain an activity, and mechanisms are resources that are used in bringing about the intended goals of the activity. Thus, for example, in an Implement Software Prototype activity, relevant controls might be such things as a high-level software design, software documentation standards, and the operating systems of the development environment. And the most salient mechanisms would likely be the programmers on the project (together with their computers). Intuitively, there are no salient inputs to this activity, as nothing is actually transformed or destroyed as the activity is actually carried out, and the output is, of course, the completed prototype.

Inputs, controls, outputs, and mechanisms are referred to generally in

IDEF0 as *concepts*, or *ICOMs* (an acronym for the four types of concept). The former term is a bit of a misnomer, for, unlike the ordinary meaning of the term, an IDEF0 concept needn't be an abstract or mental entity. Hence, because it has no connotations from ordinary language, the latter term will be used here. An ICOM, then, can be any entity — mental or physical, abstract or concrete — that plays a certain role in an activity. Note, however, that the same entity might play different roles in different activities. Thus, a particular NC machine might both be the output of a Make-NC-machine activity, and the main mechanism for transforming material input into output in a Make-widget activity. Note also that an ICOM can be a complex object (a car body, for example) that is composed of many other objects.[1]

3.2 IDEF0 Syntax: Boxes and Arrow Segments

The world according to IDEF0, then, consists of activities (functions) and ICOMs. Accordingly, the graphical language of IDEF0 contains two basic constructs: *boxes*, representing activities, and *arrow segments*, representing ICOMs. Arrow segments have a *head* — indicated explicitly by an arrowhead when necessary — and a *tail*. Arrow segments combine to form *arrows*, which will be discussed below. The basic constructs of IDEF0 are built up by connecting boxes and arrow segments together in certain allowable ways. Specifically, the head of an arrow segment can only connect to the bottom, left side, or top of a box, or to the tail of another arrow segment. The tail of an arrow segment can only connect to the right side of a box, or to the head of another arrow segment. The most basic construct of IDEF0 is depicted in a general fashion in Figure 1, along with indications of the type of entity in the IDEF0 ontology each component of the construct signifies.

Notice that the box side to which an arrow segment attaches indicates the type of ICOM that it represents *relative to the activity represented by that box*. Arrow segments representing inputs, controls, and mechanisms for the function in question attach at the head to the left side, top, and bottom of a box, respectively, and are said to be the *incoming* segments of that box. Arrow segments indicating outputs attach at the tail end to the right side of a box, and are said to be the *outgoing* segments of that box. Every box in a model must have at least one incoming control arrow segment and one outgoing output arrow segment. A control segment is required because there must be something that guides, determines, or constrains a well defined enterprise function; random, unstructured, unrepeatable activities are beyond the scope of the IDEF0 method. An output segment is required because oth-

[1] It should be noted that, when talking in general about a certain *kind* of activity, as they are wont, by an ICOM a modeler often means a corresponding *class* of particular ICOMs, e.g., the class of NC machine outputs from all Make-NC-machine activities of a certain sort. Context typically determines whether one is speaking about classes or instances, and, accordingly, we shall not be overly zealous in specifying which "level" we ourselves intend at every point in this article.

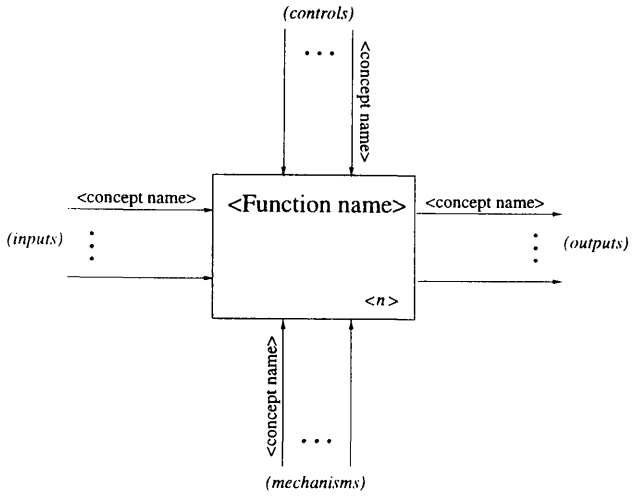

Figure 1: The Basic IDEF0 Construct

erwise there would be no purpose served by the activity, and hence it would add no value to the enterprise. Inputs, though typical, are not required, as not every function involves the consumption or transformation of some object, e.g., writing an email message. Similarly, some activities, e.g., high-level planning, may require no separate, identifiable mechanism.

3.3 IDEF0 Diagrams

Boxes and arrow segments are combined in various ways to form *diagrams*. The boxes in a diagram are connected by sequences of arrow segments, which can fork and join within a diagram as depicted in Figure 2.

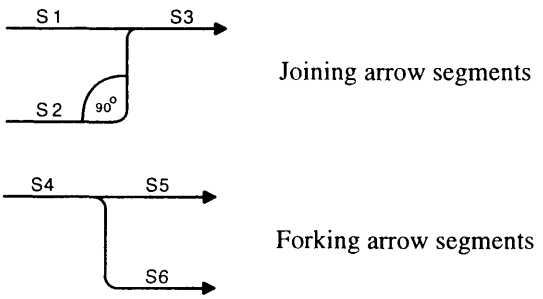

Figure 2: Arrow Segment Forking and Joining

In IDEF0, a join typically indicates either (physical or conceptual) com-

position or generalization. Hence, a (three-segment[2]) join is often said to indicate the *bundling* of two ICOMs into another, and the more complex or more general ICOM is sometimes referred to as a "bundle." Thus, in Figure 2, S1 might signify the ICOM Ad and S2 the ICOM Envelope and S3 the composite ICOM, or bundle, Mail-promo, whose instances consist of sealed envelopes containing copies of the advertisement in question. (In cases of composition there is often an underlying enterprise activity, but one which is not considered significant enough to warrant explicit representation.) Again, S1 might signify the ICOM Inventory Entry, S2 the ICOM Billing Entry, and S3 the more general, bundled ICOM Account Entries. As the term indicates, it is usually best to think of bundled ICOMs like fiber bundles in fiber-optic cables: instances of two ICOMs that are bundled together into a third are not mingled indistinguishably together, as in a confluence of two rivers; they are simply packaged together and, without losing their original characters, both delivered as inputs, controls, or mechanisms to the same functions.

A join can also simply indicate recognition of a single ICOM whose instances stem from different sources. In this case, all three segments involved in a join indicate exactly the same ICOM. Such cases are usually signified by attaching a label only to the "merged" segment (S3 in Figure 2).

A fork, naturally, is the "dual" of a join. That is, a fork indicates either (physical or conceptual) *de*composition or specialization. Forks are therefore also commonly said to indicate an *un*bundling of one ICOM into two others (one of which might be identical with the initial ICOM). As with joins, a fork can also simply indicate the recognition of a single ICOM whose instances are used as inputs, controls, or mechanism for different functions.

To illustrate, consider the diagram in Figure 3 which represents, from an accounting perspective, the activities initiated by the receipt of a customer order. As illustrated, the connected boxes in an IDEF0 diagram are represented in a "stair step" fashion (on a page or computer screen) from top left to lower right. Each box from top left to lower right is numbered sequentially beginning with 1 (with one exception, noted below); this is the box's *box number*. In the diagram in Figure 3, the two forks following the arrow segment labeled 'Fulfillment Files' indicate that the bundled ICOM Fulfillment Files includes both Customer Records that are used as controls on the Deliver function and the Price Tables and Tax Tables that serve as controls on the Bill function. Similarly, the join that merges into the arrows segment labeled

[2]Because arrow segments in standard IDEF0 syntax must be either horizontal or vertical except perhaps for 90 degree bends, forks and joins can involve no more than four arrow segments — to the join in Figure 2 one could add a segment symmetrical to S2 that joins the other three from above; analogously for the fork. Theoretically, this is no limitation, as one can get the semantic effect of an n-segment fork or join simply by means of a series of three-segment forks or joins. This semantic equivalence is one example of why one ought not to read any temporal significance into arrow segments. For example, a series of joins in a model all indicating physical compositions would not have any implications for how (instances of) the indicated ICOMs are actually composed in instances of the activity being modeled.

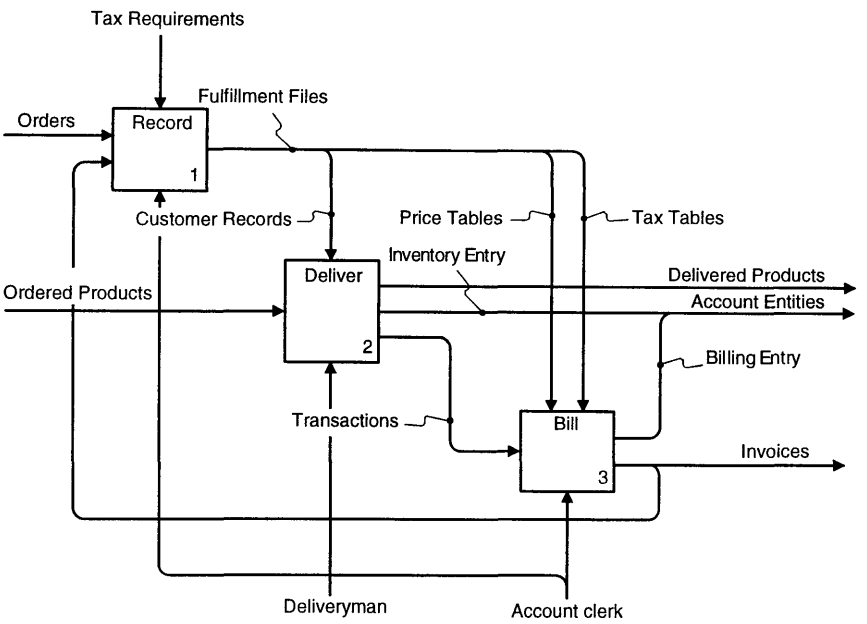

Figure 3: An IDEF0 diagram

'Account Entries' indicates that the Account Entries bundle includes both Inventory Entries and Billing Entries. The fact that the segments forking from the segment labeled 'Account Clerk' are unlabeled indicates that an Account Clerk is used as a mechanism in both the Bill and Record functions.

An *arrow* is a certain kind of sequence of arrow segments within a diagram. An arrow originating at one box and ending at another indicates a resource connection between the indicated functions — though one of those functions may be only implicit if one end of the arrow's initial or final segment is not attached to anything in the diagram. Thus, syntactically, an IDEF0 *arrow* within a diagram D is defined to be a connected sequence of arrow segments in D such that at least one end (i.e., the tail of its initial segment or the head of its final segment) is attached to a box and the other is either attached to a box or unattached to anything in the diagram. Thus, for example, in Figure 3, the Orders arrow segment is itself an arrow, as is the sequence consisting of the Fulfillment Files segment, the unlabeled segment it is attached to it (which, by convention, also signifies the Fulfillment Files bundle), and the segment labeled 'Tax Tables'. Arrows (arrow segments) that are unattached at one end are known as *boundary arrows* (*boundary arrow segments*).

3.4 IDEF0 Models

An IDEF0 *model* is a hierarchically arranged collection of IDEF0 diagrams.[3] The hierarchy is actually an (inverted) tree: there is a single root node, and each node of the tree has some finite number of "daughters"; every node except the root has only one "mother". The root node of an IDEF0 model is known as the *top-level*, or *context*, diagram of the model.[4] Unlike every other diagram in the model, the top-level diagram contains only one box. This box represents — at the coarsest granularity — the single high-level activity that is being represented by the entire IDEF0 model.

The mother-daughter relation holding between two diagrams in an IDEF0 model signifies that the daughter node is the *decomposition* of a box in the mother node. A decomposition of a box B is a diagram that represents a finer-grained view of the function signified by B. Such a diagram D is known variously as a *decomposition diagram*, *detail diagram*, or *child diagram* for B, and B is known as the *parent box* of D. Only one detail diagram per box is allowed in an IDEF0 model.

By convention, a detail diagram contains three to six boxes. The traditional justification for this is that a diagram with fewer than three boxes does not contain sufficient detail to constitute a useful decomposition; similarly, a diagram with more than six boxes contains detail that should be suppressed within that diagram and unpacked in a decomposition. Many users have found this "3-6 box rule" too constraining and have proposed replacing it with a "2-9 box rule," and in fact the latter rule has been incorporated into a proposed IEEE IDEF0 standard [IEEE97].[5]

A simple IDEF0 model for a computer assembly activity can be found in Figure 4. Each diagram within a model has a *diagram number*, and each box within a diagram a unique *node number*. The top level diagram of a model has the diagram number A-0 ("A-minus-zero") and its single box has the node number A0. The number of every other diagram is simply the node number of its parent box (as every diagram but the top-level diagram is the child diagram of some box). The node number of a box in the A0 diagram (i.e., the child diagram of the A0 box) is An, where n is box's box number within the diagram. The node number of a box within every other child diagram is simply the result of concatenating the diagram's diagram number with the box's box number. Thus, the node number of Assemble CPU is A1, while that of Install Storage Devices is A13.

Let B be a box and D a diagram within a model M. B is an *ancestor* of D (within M) just in case B is either the parent box of D (in M), or the

[3]This is not strictly correct, as an IDEF0 model is typically taken also to include textual annotations and glossary, but as the focus of this article is the graphical language proper, we have chosen to ignore these more pragmatic elements of a model.

[4]In fact the top-level diagram for a model can itself be embedded within other, "environmental" context diagrams, but this subtlety will not be discussed in this paper.

[5]At the time of this writing, this document had successfully gone to ballot, and was under revision.

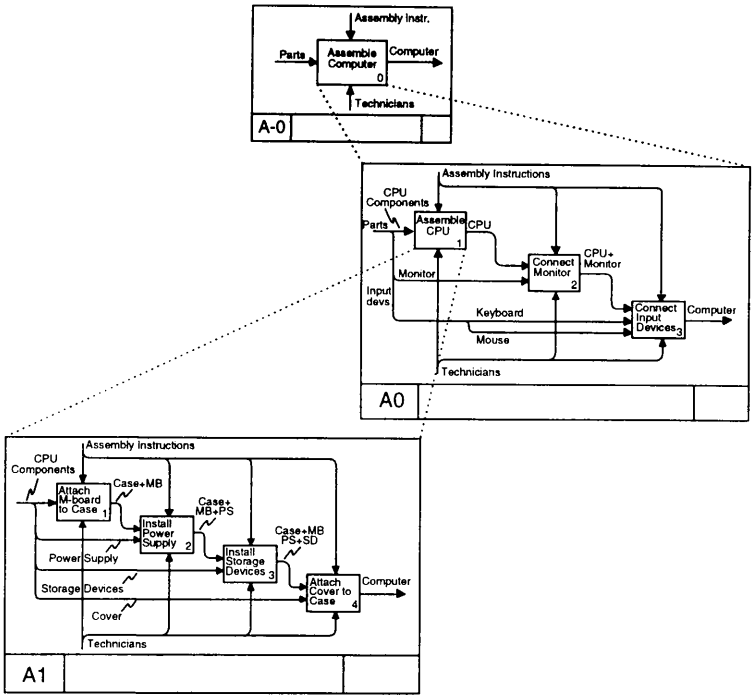

Figure 4: A Simple IDEF0 model

parent box of a diagram containing some ancestor of D (that is, just in case B is either the parent box of D, or the parent box of the diagram containing the parent box of D, or the parent box of the diagram containing the parent box of the diagram containing the parent box of D, and so on). Conversely, D is a *descendent* of B in M just in case B is an ancestor of D in M. Given this, we note that boundary arrow segments in a non-context diagram D within a model indicate ICOMs that are present in the activity indicated by some ancestor B of D — for D is simply a decomposition of B or of an activity indicated by a box in one of the descendents of B. Consequently, a boundary arrow segment that is unattached at its tail (respectively, head) can be correlated with an incoming (respectively, outgoing) arrow segment for some ancestor of D. Such a correlation is typically accomplished by labeling both arrows with the same name.[6] Conversely, and more strongly, every incoming or outgoing segment of a box with descendents *should* be correlated with an appropriate boundary arrow segment in one of its descendents (else

[6]Traditionally, IDEF0 has used a somewhat awkward system of "ICOM codes" to achieve such correlations. However, ICOM codes are both unnecessary, as the same effect can be achieved by the consistent use of names, and are also largely rendered otiose by modern modeling support software which can track such correlations with ease.

the exact function of the indicated ICOM must not be clear).

If an arrow segment S attached to a parent box is correlated with a boundary segment S′ that is not in its child diagram, then S is said to be *tunneled downwards*, and the arrow segment S′ with which it is correlated is said to be *tunneled upwards*. Tunneling simply provides a mechanism for "hiding" the role of a given ICOM in a function through the successive decompositions of the box representing that function until the appropriate level of granularity is reached.

4 The IDEF1X Data Modeling Method

Just as IDEF0 introduces a specialized ontology tailored for capturing business activities, and a specialized language for building models of those activities in terms of that ontology, so IDEF1X introduces a specialized ontology and a corresponding language tailored to build database models. We begin with a discussion of its ontology.

4.1 The IDEF1X Ontology: Entities, Attributes, and Relationships

Not surprisingly, the ontology of IDEF1X corresponds closely to the ontologies other database modeling languages such as the Entity-Relationship (ER) and NIAM modeling languages. The basic ontological categories of IDEF1X are *entities*, *attributes*, and *relationships*. We discuss each category in turn.[7]

4.1.1 Entities

Entities are simply classes of actual or — when "to be" situations are being modeled — possible things in the world. Entities can comprise concrete objects, such as employees and NC machines; more idealized objects such as companies and countries; and even abstract objects like laws and space-time coordinates. The things comprised by a given entity are known as the *members* or *instances* of the entity. IDEF1X entities thus correspond to ERA entity sets and NIAM entity classes.[8]

[7]It should be noted that we will only be discussing so-called "key-based" views. Officially, IDEF1X models can contain numerous "views", where a view, like the notion of a model here, is a structured collection of entity boxes and relationship links. Views differ in the constraints they satisfy. Specifically, the ER view does not require the identification of keys, and allows "nonspecific", many-to-many relationships (see below for definition of these notions). For the sake of brevity, in this paper we are identifying models with what are known as "fully-attributed" views in IDEF1X, in which keys must be identified and all many-to-many relationships must be resolved into functional relationships.

[8]The term 'entity' is rather unfortunate, since in ordinary language it is a rough synonym for 'thing' or 'object', i.e., for individual *instances* of classes rather than classes themselves. IDEF1 uses the more appropriate term 'entity class'.

4.1.2 Attributes

Every entity has an associated set of attributes. Attributes are simply functions, or mappings, in the mathematical sense: an attribute associates each instance of a given entity with a unique value. An attribute α is *for* a given entity E if it is defined on all and only the instances of E.[9] In IDEF1X, the set of values that an attribute can return is known as the attribute's *domain.*[10] The domain of every attribute referred to in an IDEF1X model is always one of several familiar data types; specifically, it is either the type *string*, a numerical type of some ilk, the type *boolean*, or else a subtype of one of these basic types. So, for example, common attributes for an EMPLOYEE entity might be Name (of type *string*), Citizenship (subtype of *string*, viz., names of countries), Yearly-salary (*positive integer*), Marital-status (*boolean*), and so on.

A central notion in IDEF1X is that of a *candidate key*, or simply, *key*. A key for an entity E is a set of attributes for E that jointly distinguish every instance of the entity from every other. More exactly, where α is an attribute, let $\alpha(e)$ be the value of α applied to e. Let A be a set of attributes for an entity E. Then A is a key for E just in case, for any distinct instances e, e' of E, there is an attribute $\alpha \in A$ such that $\alpha(e) \neq \alpha(e')$. Ideally, a key should be a *smallest* set of this sort, in the sense that no proper subset of a key is also a key. If an attribute α is a member of a key, it is said to be a *key attribute.*

4.1.3 Relationships

Relationships are classes of associations between instances of two (possibly identical) entities. In the context of IDEF1X, one of the two entities is identified as the *parent* entity and the other as the *child* entity. Let R be a relationship, and let E_P^R be its parent entity and E_C^R its child. Then the one general requirement on relationships is that for each instance e of E_C^R there is at most one instance e' of E_P^R such that e is associated (by R) with e'.[11] Also, typically, in an IDEF1X model, no instance of a relationship's child entity fails to be associated with an instance of its parent, though this is not always required. (See the notion of an "optional" non-identifying relationship below.) If E is the child of a relationship R and E' the parent, then R will be said to *link E to E'*. (This is not standard IDEF1X terminology, but it proves very useful for exposition.)

[9] Partial attributes — i.e., attributes that are not defined on all the instances of an entity — are allowed in ER views.

[10] This is another unfortunate choice of terminology, as the term 'domain' in mathematics is the usual name for the set of *arguments* for a function, and the term 'range' denotes the set of its possible values, i.e., the attribute's "domain" in the sense of IDEF1X.

[11] In ER views, "non-specific" relationships are allowed that don't satisfy this requirement; specifically, in a non-specific relationship an instance of the child might be associated with more than one instance of the parent.

It is convenient to think of a relationship R as a class of ordered pairs $\langle a, b\rangle$ such that the first element a of each such pair is an instance of R's child entity and the second element b is an instance of its parent entity. The general requirement on relationships, then, can be expressed simply as the requirement that a relationship R be *functional*, in the sense that, for $a \in E_C^R$ and $b \in E_P^R$, if Rab (i.e., if $\langle a, b\rangle \in R$) and Rac, then $b = c$. Given this, to say that a given instance e of R's child entity E_C^R is *associated (by R) with* an instance e' of R's parent entity E_P^R is simply to say that eRe'; likewise, to say that a given instance e of E_P^R is associated with an instance e' of E_C^R is to say that $Re'e$. We say that R is *total* if for each instance e of E_C^R there is an instance e' of E_P^R such that Ree'. Otherwise R is said to be *partial*. Since relationships R are functional, we will sometimes write '$R(a)$' to indicate the unique object b such that Rab (when it is known that there is such an object b).

4.1.3.1 Cardinality The *cardinality* of a relationship R signifies how many instances of the child entity a given instance of the parent is associated with. Often a relationship has no specific cardinality; one instance of the parent might be associated by R with two instances of the child, another with seventeen. The most that can be said in such cases is that the relationship has a cardinality of *zero, one, or more*, which is true under any circumstances. But often enough this is not the case. IDEF1X marks out in particular the following cardinalities for R: *one or more* (signifying that R, viewed as a function from E_C^R to E_P^R, is onto, or surjective); *zero or one* (indicating that R, viewed as a function, is one-to-one, or injective); *exactly n*; and *from n to m*.

4.1.3.2 Attribute Migration The functionality of relationships leads to the important notion of key attribute *migration*. Suppose R links E to E' and let α be a key attribute for the parent entity E'. Because R maps each instance e of E to a unique instance e' of E', a new (migrated) attribute for E can be defined as the *composition* $R \circ \alpha$ of α and R.[12] Thus, more procedurally, to discover the value $R \circ \alpha(e)$ of the migrated attribute $R \circ \alpha$ on a given instance e of E, one first finds the instance e' of E' associated with e by R, and then applies α to e'; i.e., $R \circ \alpha(e) = \alpha(R(e))$. This is the value of the migrated attribute on e. (An example is given below.)

The notion of migration is often documented misleadingly so as to suggest that a migrated attribute in the child entity of a relationship is the very same attribute as the migrating attribute in the parent entity. Since they are attributes for different entities, however, the two must be distinct. It is more correct to characterize migration as a *relation* involving two attributes and a relationship. More exactly, let R be a relationship and α and α' attributes, and let E be the child entity of R and E' the parent entity. Then we say that

[12] Where, as usual, $f \circ g(x) = g(f(x))$.

α' *migrates from* E' *to* E *as* α *via* R if and only if (i) α and α' are attributes for E and E', respectively, (ii) R links E to E' and (iii) for all instances e of E, $\alpha(e) = \alpha'(R(e))$. We will call α' the *migrating* attribute and α the *migrated* attribute (relative to R). Note that a migrated attribute relative to one relationship can itself be a migrating attribute relative another.

4.1.3.3 Categorization Relationships A particularly important type of relationship in IDEF1X is a *categorization relationship*. Basically, a categorization relationship is just the identity relation restricted to a certain subclass of a given entity; that is, a categorization relation maps a member of a subclass of a given entity to itself in that entity. The importance of these relationships is that they are used to form *categorization clusters*, which divide a given entity — known as the *generic* entity in the cluster — into several disjoint subclasses or *category* entities. Thus, the generic entity in a cluster might be the entity EMPLOYEE, and SALARIED_EMPLOYEE and HOURLY_EMPLOYEE the category entities in the cluster. A category cluster is *complete* if the category entities jointly constitute a *partition* of the generic entity, i.e., if every instance of the category entity is an instance of a (unique) category entity.

It is often useful to identify a *discriminator attribute* for a category cluster that returns, for each instance of the generic entity, a standard name for its category. Thus, the discriminator attribute for the EMPLOYEE cluster above would return either the string 'SALARIED_EMPLOYEE' or 'HOURLY_EMPLOYEE' on each generic entity instance. (For incomplete clusters, a discriminator attribute would have to be either undefined on generic instances that are in no category, or else would have to return a string indicating this, e.g., 'NIL'.)

4.2 The IDEF1X Language and its Semantics

Entities, attributes, and relationships constitute the basic ontology of IDEF1X the basic categories of things that one talks about in the IDEF1X language. In this section we describe the language itself and its semantical connections to these objects.

The basic syntactic elements of the IDEF1X language are *entity boxes*, *attribute names*, and various kinds of *relationship links*. These elements, of course, signify entities, attributes, and relationships, respectively. An IDEF1X *model* is a collection of entity boxes, attribute names, and relationship links that satisfy certain conditions, which we will state in the course of our exposition. As with our account of IDEF0, then, we will continue to use the term 'model' to indicate a certain kind of complex syntactic entity. However, an entity, attribute, or relationship can be said to be "in" a model insofar as that entity, attribute, or relationship is indicated by a corresponding entity box, attribute name, or relationship link in the model.

Entity boxes come in two varieties, ones with square corners and ones with rounded corners, as indicated in Figure 5.

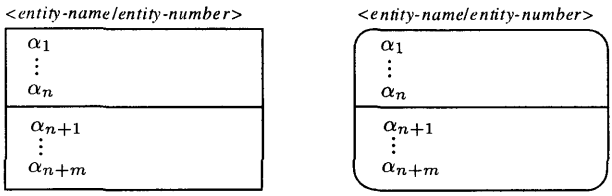

Figure 5: Entity Boxes

The α_i are attribute names. The names $\alpha_1, ..., \alpha_n$, written above the line, indicate the members of a distinguished key for the indicated entity, known (in the context of a model containing the given entity box) as the *primary key* for the entity. n here must be at least 1; that is, it is required that a primary key be identified for every entity indicated in a model. The same entity, of course, could have a different primary key in a different model, although, of course, it would have to be denoted by a correspondingly different entity box in that model. $\alpha_{n+1}, ..., \alpha_{n+m}$ indicate other, non-key attributes for the entity.

Which of the two kinds of box to use for an entity in a model depends on the kinds of relationships that link that entity to other entities indicated in the model. Perhaps the most common type of relationship between entities in a model is an *identifying* relationship, the IDEF1X syntax for which is given in Figure 6. To define this notion, note first that it is a requirement on IDEF1X models that, for any relationship R, all and only the attributes in the primary key of R's parent entity migrate to its child entity via R.[13] R is an *identifying* relationship if all of the attributes in the parent's primary key migrate via R as attributes in the child's primary key; otherwise R is a *nonidentifying* relationship. The idea here is that, procedurally, an instance e of the child entity in a relationship can be identified — i.e., its key attribute values determined — only by first identifying e's associated instance e' in the parent entity, i.e., by first determining all of *its* (e''s) key attribute values. If an entity E is the child entity in an identifying relationship R in a model, then a box with rounded corners is used to indicate E in that model. Otherwise, a box with square corners is used.

A simple example is given in Figure 7. In this example, the primary key attribute Dept_number migrates as the attribute Works_in.Dept_number, which appears as a primary key attribute of EMPLOYEE. The relationship is therefore, by definition, an identifying one. In the example, Emp_numbers alone are not in general sufficient to distinguish one EMPLOYEE from an-

[13]Migrated attributes are sometimes referred to as "foreign keys", or, a bit less problematically, "foreign key attributes", and are often marked with the expression '(FK)'. This marking is otiose if the full name of the migrated attribute is given (i.e., if a role name is used in naming the attribute; see below) but can be heuristically useful if role names are suppressed.

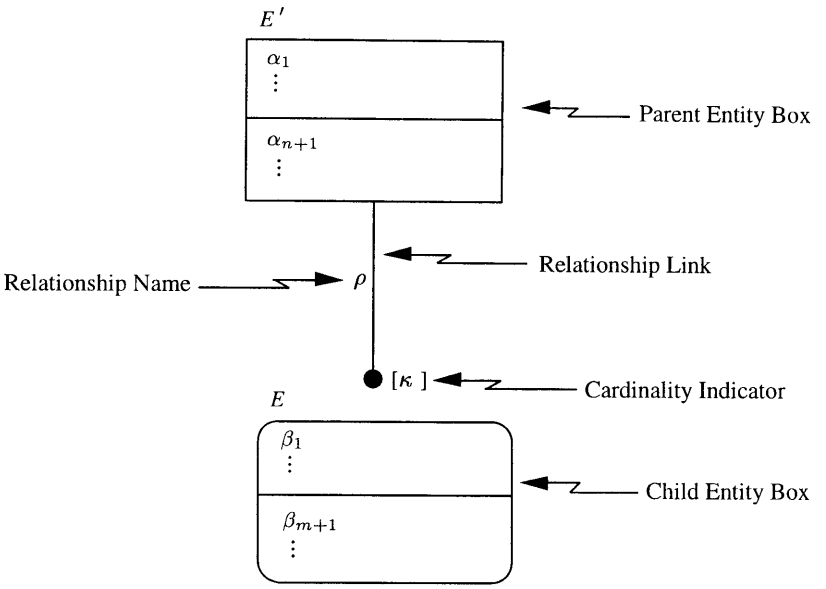

Figure 6: Syntax for Identifying Relationships

other; Emp_numbers are unique only within DEPARTMENTs. Hence, one must also know the Dept_number of the DEPARTMENT in which an EMPLOYEE works to distinguish him or her from every other EMPLOYEE. Hence, the primary key for EMPLOYEE also contains the migrated attribute Works_in.Dept_number. Note that the relationship name 'Works_in', or some related identifier (known in the context as a "role name"), becomes part of the name of the migrated attribute. This is to indicate the relationship relative to which the migration has occurred. By convention, if there is no possibility of confusion, the very same name is used for the migrated attribute. Thus, because there is no such possibility in the example (since there is only one relationship linking EMPLOYEE to DEPARTMENT), 'Dept_number' could have been used in both entity boxes. An attribute like Dept_number or SSN that is not migrated relative to any relationship in the model is said (relative to that model) to be *owned* by the entity it is defined on.

One further construct in Figure 6 requires comment, viz., the cardinality indicator κ. This marker, of course, indicates the cardinality of the relation. The brackets around κ signify that cardinality indicators are optional. If no indicator is present, then the relationship in question can have any cardinality. 'P', by contrast, indicates the relationship is many-to-one; 'Z' that it is one to zero or one; a specific numeral ν indicates that the cardinality is exactly n, where ν denotes n; and ν-μ indicates a cardinality of n to m, where ν and μ denote n and m, respectively.

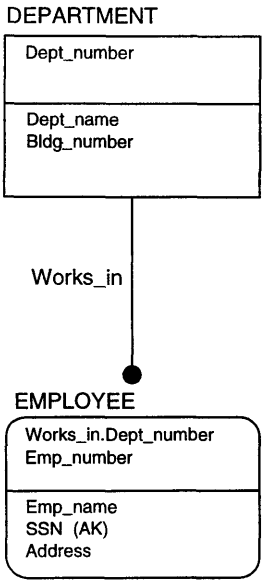

Figure 7: Example of an Identifying Relationship

As noted above, if R is not an identifying relationship (and no identifying relationship links E_C^R to any other entity in the model), then a square-cornered box is used to indicate the child entity. A dashed line rather than a solid line is used to indicate non-identifying relationships. A non-identifying relationship R is said to be *mandatory* if R is a total function from E_C^R to E_P^R, i.e., if every instance of R's child entity is associated with an instance of R's parent entity; otherwise R is said to be *optional.* For example, let E' be a class of offices in a business and let E be the class of computers that exist in the business, and let R be the Located-in relationship. Most, but perhaps not all, computers will be located in offices, but some might, e.g., have been sent out for repair, and hence are not located in any office. If this can be the case, then Located-in is an optional relationship.[14]

An optional relationship is indicated by a dashed line with a small diamond at the parent end of the link, as shown in Figure 8.

[14]Strictly speaking, the difference between mandatory and optional relationships really applies more accurately to the labeled relationship *links* in a model. Entities, attributes and relationships form what in mathematical logic are known as *interpretation* of the basic syntactic elements of IDEF1X. An interpretation can be said to *validate* an IDEF1X model if its entities, attributes, and relationships comport with the constraints expressed in the model (e.g., if the relationship associated with a one-to-n link really is one-n). To call a relationship link mandatory, then, is to say that it can only be associated semantically in any interpretation with a relationship that is a total function. The interested reader is referred to [End72].

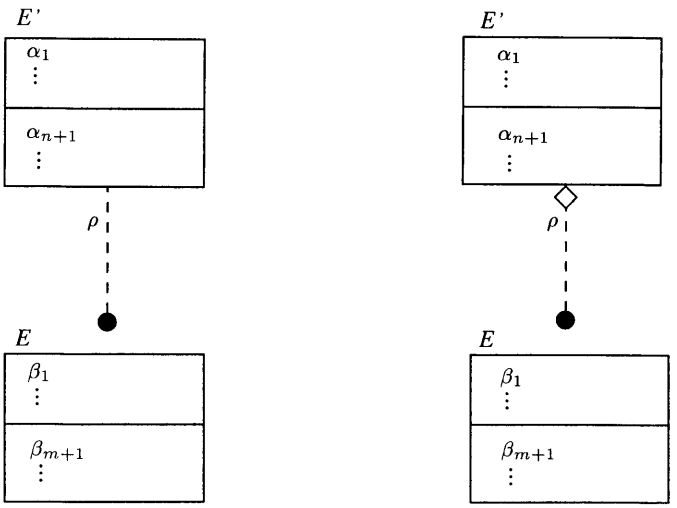

Figure 8: Syntax for Non-identifying Relationships

Any subset of an entity's attributes in a model that constitute a further key is known as an *alternate* key for the entity (relative to that model). The names of members of an alternate key are marked with the string '(AK)', as illustrated by the attribute SSN in Figure 7. Should there be more than one alternate key, then the keys are ordered (arbitrarily) and the names of the attributes in the first key are marked with the string '(AK1)', those in the second with '(AK2)', and so on. (It is possible, but uncommon, that the same attribute be in different alternate keys, and hence for an attribute name to be marked by more than one of the terms '(AKn)').

Finally, the syntax for a complete categorization cluster with three category entities is exhibited in Figure 9. A name for the discriminator attribute is written alongside the circle beneath the generic entity box. In general, clusters with n category entities are represented with n relationship links running from the lower of the two horizontal lines beneath the circle to n entity boxes. Note that the names of the primary key attributes for every category entity are identical with their counterparts in the generic entity. This reflects the fact, noted previously, that the relationship linking a category entity to its generic entity is the identity relation. Hence, each key attribute in the generic entity migrates to each category entity as a restricted version of itself that is defined only on those instances of the generic entity that are instances of the category entity. This "near identity" of the migrating and migrated attributes warrants using the same attribute name in the boxes for both generic and category entities.

Incomplete categorization relationships are indicated in precisely the same

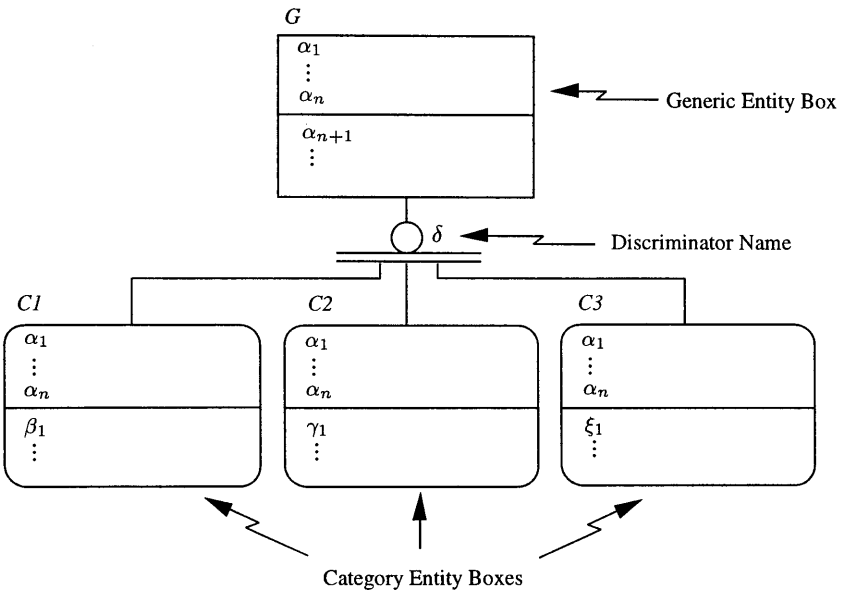

Figure 9: Complete Categorization Cluster Syntax

way, except that a single rather than a double horizontal line is used beneath the circle.

5 The IDEF3 Process Modeling Method

The IDEF3 modeling method is used to construct models of general enterprise processes. Like IDEF0 and IDEF1X, it has a specialized ontology and, of course, a corresponding language, which we detail in the following sections.

5.1 The IDEF3 Ontology: UOBs, Objects, and Intervals

Because the terms 'process' and 'activity' are rough synonyms in ordinary language, one might wonder what distinguishes the subject matter of IDEF0 from that of IDEF3. In one sense, nothing; both are concerned with the modeling of actual and possible situations. The difference is a matter of focus: features of situations that are essential to IDEF0 activities are generally ignored in IDEF3; and, conversely, features essential to IDEF3 processes are ignored in IDEF0. More specifically, because IDEF0 is concerned primarily with the ways in which business activities are defined and connected by their products and resources, IDEF0 activities are characterized first and foremost

in terms of their associated inputs, outputs, controls and mechanisms. By contrast, because IDEF3 is intended to be a general process modeling method without, in particular, a specific focus on products and resources, an IDEF3 process — also known as a *unit of behavior*, or *UOB*, to avoid the connotations of more familiar terms — is characterized simply in terms of the *objects* it may contain, the *interval of time* over which it occurs, and the *temporal relations* it may bear to other processes. Thus, IDEF0 (by default) ignores the temporal properties of situations (in particular, it is not assumed that an activity must occur over a continuous interval), and it highlights certain roles that objects play in them. By contrast, IDEF3 (by default) ignores those roles and simply records general information about objects in situations and the temporal properties of, and relations among, situations. IDEF3 is thus particularly well-suited to the construction of models of general enterprise processes in which the timing and sequencing of the events in a process is especially critical. Notably, it is a particularly useful language to use in the design of complex simulation models.

5.2 The IDEF3 Language and its Semantics

The basic elements of the IDEF3 lexicon for building process models are illustrated in Figure 10. UOB boxes, of course, in the context of an IDEF3

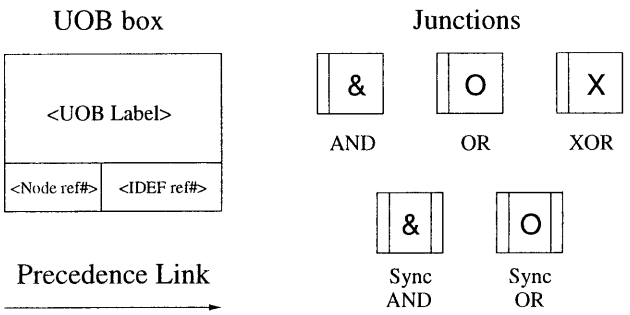

Figure 10: The Basic IDEF3 Process Description Lexicon

model, signify UOBs, and precedence links signify a certain kind of temporal constraint. Every UOB box has an associated *elaboration*, i.e., a set of logical conditions, or constraints, written either in English or, more ideally, in a formal logical language. A UOB box can signify a given UOB A only if the latter satisfies the logical constraints in the elaboration of the former. In such a case we say that A is an *instance* of the UOB box. Junctions, too, can have elaborations.

5.2.1 Syntax for the Basic IDEF3 Construct

The basic construct of IDEF3 is illustrated in Figure 11. Box 1, with the

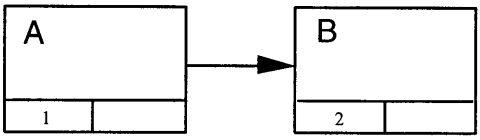

Figure 11: The Basic IDEF3 Construct

label 'A' at the "back" end of the link is known as the *source* of the link and box 2 with label 'B' at the "front" end of the link is known as the *destination* of the link. If Figure 11 is considered as a complete IDEF3 model, box 1 is known as the (immediate) predecessor of box 2 in the model, and box 2 the (immediate) successor of box 1. The '1' in box 1 and the '2' in box 2 are the *node reference numbers* of the boxes, and are assumed to be unique within a model. The corresponding area to the right of the node reference number in a UOB box is optionally filled by an *IDEF reference number*, a broader identifier for the purpose of locating that model element with respect to numerous IDEF models.

5.2.2 Semantics for the Basic Construct

The meaning of an IDEF3 model is best understood in terms of its possible *activations*, the possible real world situations that exhibit the structure specified in the model. In the simplest case, an activation of a model is a collection of UOBs that satisfy the temporal constraints exhibited by the structure of the precedence links in the model. In general, there are many different patterns of activation for a given model. However, there is only one possible activation pattern for simple two box models like Figure 11, viz., when a single UOB A of the sort specified in the box 1 is followed by a UOB B of the sort specified in box 2. More precisely, a legitimate activation of Figure 11 as it stands is any pair of situations A and B that are instances of boxes 1 and 2, respectively, and where B does not start before A finishes.

5.2.3 Junctions

Junctions in IDEF3 provide a mechanism to specify the logic of process branching. Additionally, junctions simplify the capture of timing and sequencing relationships between multiple process paths.

5.2.3.1 Junction Types An IDEF3 model can be thought of as a general description of a class of complex processes, viz., the class of its activations. Such a description is rarely linear, in the sense that the processes it picks out

always exhibit the same linear pattern of subprocesses. More typically, they involve any or all of four general sorts of "branch points:"

1. Points at which a process satisfying the description diverges into multiple *parallel* subprocesses;

2. Points at which processes satisfying the description can differ in the way they diverge into multiple (possibly nonexclusive) *alternative* subprocesses;

3. Points at which multiple parallel subprocesses in a process satisfying the description converge into a single "thread;" and

4. Points at which processes satisfying the description that had diverged into alternative subprocesses once again exhibit similar threads.

IDEF3 introduces four general types of junction to express the four general sorts of branch points. The first two sorts are expressed by "fan-out" junctions: Conjunctive fan-out junctions represent points of divergence involving multiple parallel subprocesses, while disjunctive fan-out junctions represent points of divergence involving multiple alternative subprocesses. The last two sorts of branch point are expressed by "fan-in" junctions: conjunctive fan-in junctions represent points of convergence involving multiple parallel subprocesses, while disjunctive fan-in junctions represent points of convergence involving multiple alternative subprocesses. There is one type of conjunctive, or AND, junction, indicated by '&'. There are two types of disjunctive junction: inclusive and exclusive junctions, or OR and XOR junctions, respectively, depending on whether the alternatives in question are mutually exclusive. OR junctions are indicated by an 'O', and XOR junctions by an 'X'.

Junction syntax is illustrated in Figure 12, where γ is either '&', 'O', or 'X'. Although this figure shows only two UOB boxes to the right of a fan-out junction and to the left of a fan-in, arbitrarily many are permitted in an IDEF3 model in general.

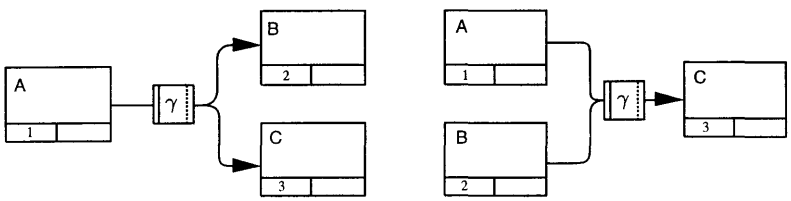

Figure 12: Junction Syntax

5.2.3.2 Junction Semantics The intuitive meaning of junctions is straigh forward. It will be enough to use Figure 12. Letting γ be '&' in the figure, an activation of the model on the left will consist of an instance A of box 1 followed by instances B and C of boxes 2 and 3. If the junction is synchronous, then B and C will begin simultaneously. (Note in particular that, for nonsynchronous junctions, there are no constraints whatever on the temporal relation between B and C; all that is required is that both occur after A.) Similarly, an activation of the right model in the figure will consist of instances A and B of boxes 1 and 2 followed by a single instance C of box 3; and if the junction is synchronous, then, A and B will end simultaneously.

For OR (XOR) junctions, if γ is 'O' ('X'), then an activation of the model on the left in the figure will consist of an instance A of box 1 followed by either an instance B of box 2 or an instance C of box 3 (but, for XOR junctions, not both). If the OR junction is synchronous, then, should there be instances of both boxes 2 and 3, they will begin simultaneously. Similarly, an activation of the right model in the figure will consist of an instance of either box 1 or box 2 (but, for XOR junctions, not both) followed by an instance of box 3. If the OR junction is synchronous, then, should there be instances of both boxes 1 and 2, they will end simultaneously.

These semantic rules generalize directly, of course, for junctions involving arbitrarily many UOB boxes. Control conditions on branching and concurrency on a class of processes — e.g., the conditions that determine which of two paths to follow at an XOR junction — are often placed in the elaboration of a junction.

5.3 Models and Schematics

An IDEF3 model is a collection of one or more IDEF3 *process schematics*, which are built from UOB boxes, precedence links, and junctions in natural ways. Intuitively, a schematic is simply a single "page" of a model, a view of (perhaps only a part of) a process from a given perspective at a single uniform granularity.

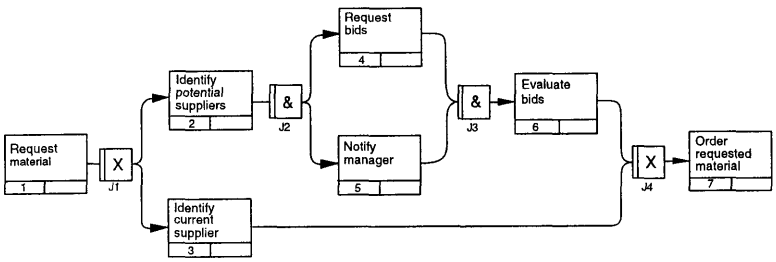

Figure 13: A Small IDEF3 Schematic

A simple example of a schematic is seen in Figure 13. In this schematic, a request for material is followed by either the identification of the current supplier or the identification of potential suppliers. (A condition attached to the junction might indicate that the latter path is taken only if there is no current supplier; but this common sense condition, of course, cannot be derived from the bare semantics of the language alone.) If a current supplier is identified then an order is placed. Otherwise, the identification of potential suppliers is followed by both a report to the manager and a request for bids from the potential suppliers. When both of these tasks are complete, the bids that have arrived are evaluated and an order placed to the winning bidder.[15]

The formal syntax for IDEF3 process schematics is rather laissez-faire; the onus is on the modeler to construct coherent models, i.e., models with possible activations. However, although basically straightforward, the syntax requires more mathematical apparatus than is appropriate here to specify precisely. Informally, though, there are essentially two main rules:

1. A UOB box can be the source or destination of no more than than one precedence link; and

2. A schematic must contain no loops.

The motivation behind the first rule is that precedence links with the same box as source or destination would indicate a point at which there are parallel subprocesses diverging or converging, or a point at which alternative subprocesses can be seen to diverge or converge across different processes satisfying the description. The purpose of fan-out and fan-in junctions is to indicate just such points in a description meant to capture the general structure exhibited by many possible processes.

Regarding the second rule, a *path* through a schematic is a sequence of UOB boxes, junctions, and precedence links such that each element of the sequence (save the last, if there is a last element) is connected to its successor. A loop, or *cycle*, in a schematic is a path in the schematic whose first element is identical to its last. At first blush, the second rule might seem highly undesirable, as loops appear to be very common structural features of many processes. Consider, for example, the process depicted in Figure 14 (in apparent violation of Rule 2).

The problem with loops is that they are inconsistent with the semantics of the precedence link. As noted above, the precedence link indicates temporal precedence. This relation is *transitive*, that is, if UOB A is before B in time, and B before C, then A is before C as well. Given that, suppose box b1 is linked to box b2, and b2 to b3 in a model M, and that A, B, and C are instances of b1, b2, and b3, respectively, in some activation of M. By the basic semantics of the precedence link, A must precede B and B must precede C. But then, by the transitivity of temporal precedence, A must precede C. Now,

[15] Henceforth, junction numbers will be suppressed.

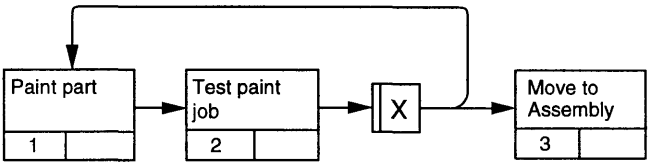

Figure 14: Process with an Apparent Loop

notice that, on this understanding of the precedence link, a loop in a model would mean that one point in an activation of the model — one point in a possible or actual process — could return to an earlier point, and hence that the later point could precede the earlier point. Clearly, though, given the direction of "time's arrow," this is not possible; the past remains ineluctably past and inaccessible; once past, no point in time can be revisited.

Why then is there a temptation to use loops in process models? The answer is clear; in some processes — the one depicted in Figure 14, for instance — a particular *pattern* is instantiated many times. It is therefore convenient and, often, natural simply to indicate this by reusing that part of a model that represents the first occurrence of this pattern, rather than iterating separate instances of it. As noted, though, this is not compatible with the general semantics of the precedence link. Strictly speaking, then, loops must be "unfolded" into noncycling structures. If there is a bound on the number of iterations, the corresponding noncycling model will be finite. Otherwise it will be infinite; the infinite unfolded model corresponding to Figure 14 is exhibited elliptically in Figure 15.

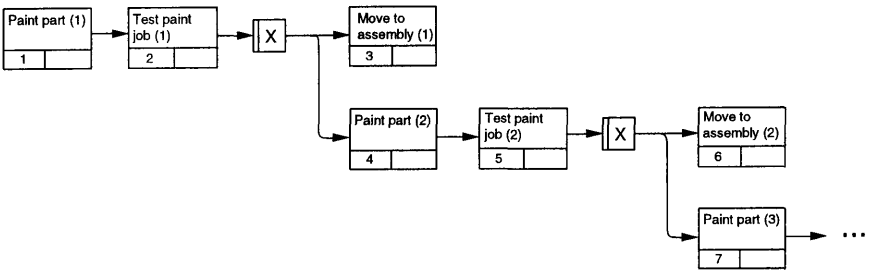

Figure 15: Unfolded Model of the Process Depicted in Figure 14

That noted, it has already been acknowledged that models with loops are often convenient and natural. Indeed, given the ubiquity of processes with iterated patterns, to require modelers explicitly to unfold loops in general would rob IDEF3 of a significant degree of its usability. Consequently, IDEF3 allows models with loops — however, importantly, these are under-

stood syntactically not as primitive constructs but as *macros* for their unfolded counterparts. So understood, loops are semantically innocuous and can be used without qualms.

5.3.1 Referents

Loops are typically indicated in IDEF3 by means of *referents* in process models. Referents are theoretically dispensable, but are useful for reducing clutter. In the context of a process model, referents are used to refer to previously defined UOBs. Referents therefore enhance reuse, as one can simply refer to the indicated schematic or UOB box without explicitly copying it into the referring model.

Referents come in two varieties: *call-and-wait* and *call-and-continue.* Their syntax is seen in Figure 16. The referent type of a referent can be either

Call and Continue Referent Call and Wait Referent

Figure 16: Referent Syntax

'UOB', 'SCENARIO', 'TS', or 'GOTO'. A UOB referent points to a previously defined UOB box, a scenario referent points to a model ('scenario' is the name for the complex UOB described by a model), a TS referent points to an object state transition schematic (see below), and a GOTO points to a UOB box or model. A GOTO referent indicates a change of process control to a UOB or scenario indicated by the referenced UOB box, model, or junction. In each case, the *locator* in a referent specifies the (unique) reference number of the UOB, scenario, or state transition in question. Referents, too, have associated elaborations.

As the names suggest, a call-and-wait referent calls a particular UOB or transition, and execution of the calling model halts until the called UOB or transition completes. By contrast, a call-and-continue referent simply calls a UOB or transition without any halt in the execution of the calling model. Typically, in IDEF3, a GOTO referent, rather than a backward-pointing precedence link, is used to express looping;[16] thus, on this approach, the process intended by Figure 14 would be captured as in Figure 17. Use of precedence links to express looping, however, is permitted.

[16]More than anything, perhaps, this simply reflects the way most IDEF3 support software works.

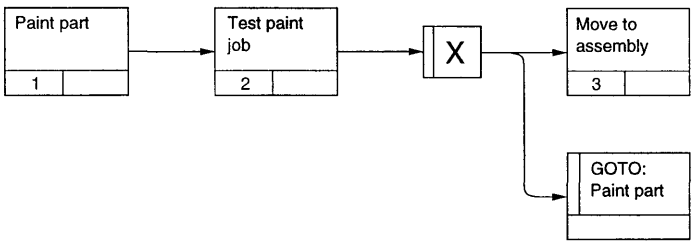

Figure 17: Looping with a GOTO Referent

5.3.2 Decompositions

A *decomposition* of a UOB box in a model is simply another IDEF3 schematic, one that purports to provide a "finer-grained" perspective on the UOB signified by the box. In a fully-fledged IDEF3 model, each schematic is either the decomposition of a UOB box in some other schematic, or else is the unique "top-level" schematic which is not the decomposition of any other schematic. That a given box in a schematic in a model has a decomposition in the model is indicated by shading, as illustrated in Figure 18.

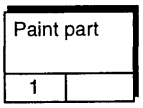

Figure 18: Decomposition Syntax

5.4 Object State Transition Schematics

Initially, process schematics were the only part of the IDEF3 language. However, it soon became apparent that modelers often desired to take "object-centered" views of processes, views that focus not so much on the situations that constitute a process, but on the series of states that certain objects within those processes pass through as the process evolves. This led to the addition of *object state transition schematics*, or simply *transition schematics* to the IDEF3 language.

5.4.1 Syntax for Basic Transition Schematics

The basic lexicon for transition schematics is shown in Figure 19.

As can be seen, the label for a state symbol displays the name of a state and, optionally, the name of the general kind of thing that is in the state. For example, the state of being hot might be labeled simply by means of

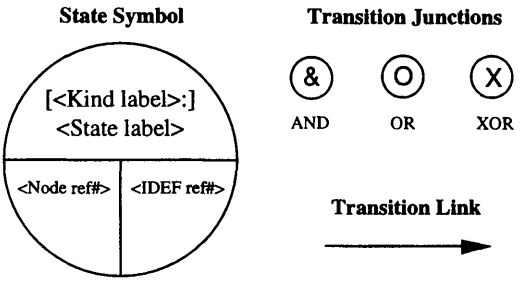

Figure 19: Lexicon for State Transition Schematics

the label HOT. If it is hot *water* in particular, though, and that fact is relevant, then the more complex label WATER:HOT could be used. (Node references and IDEF numbers in state symbols have the same role as in process schematics, and will be suppressed in the examples to follow.) An arrow (indistinguishable from a precedence link), known as a *transition link*, is used to indicate a transition from one state to another, as illustrated in Figure 20. 'K1' and 'K2' indicate optional kind (class) names, and 'S1' and 'S2' names for states.

Figure 20: Basic Transition Schematic Syntax

5.4.2 Semantics for Basic Transition Schematics

In general, the semantics of a basic transition schematic is simply that, in an occurrence of the indicated transition, there is first an object x (of kind K1) in state S1, and subsequently an object y (of kind K2) that comes to be in state S2; that is, to have an instance of the transition schematic in question, it is required that x be in state S1 before y comes to be in state S2. It is permitted, though perhaps not typical, that $x \neq y$; and it is permitted, though perhaps not typical, that x remain in state S1 after y comes to be in state S2.

It is important to note that, despite having the same appearance, the semantics of the arrow of transition schematics is somewhat different than the semantics of the precedence link. The precedence link implies full temporal

precedence: in an activation of a simple precedence connection, an instance of the UOB box at the tail of the link must end no later than the point at which an instance of the UOB box at the head of the link begins. By contrast, in an object schematic, the arrow implies precedence only with regard to starting points: the object that is in the state indicated at the tail of the arrow must be in that state before the transition to an object in the state indicated at the head of the arrow. The reason for this weaker sort of precedence in state transition schematics is simply the point noted in the previous paragraph: a transition only involves a change from an object in one state to an object (possibly the same object, possibly different) in another; though it may not be typical, the object in the initial state of the transition needn't cease being in that state after the transition. To allow for this type of transition, the weaker semantics is used for the arrow in object transition schematics. There is no potential for confusion, however, as the meaning of the arrow remains constant within each type of schematic.

5.4.3 Using UOB Referents in Transition Schematics

Because (in the context of process modeling) objects are in states within UOBs, and because transitions occur inside UOBs, it is useful and informative to be able to record information about related UOBs in a transition schematic. This is accomplished by attaching UOB referents to various parts of a transition schematic. The most common use of UOB referents is to attach them to the arrow in a transition schematic, as illustrated in Figure 21.

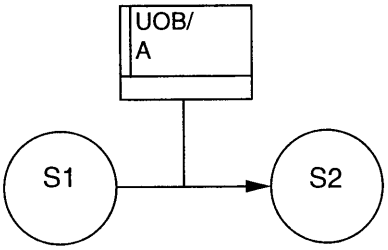

Figure 21: Use of a UOB Referent in a Transition Schematic

The default semantics here is fairly weak. The figure signifies only that in transitions of the indicated sort there will be an object x in state S1 prior to or at the start of a UOB A (satisfying the conditions specified in the referent), and subsequently an object y at some point after the beginning of A. Stronger conditions — e.g., that x=y, that x and y occur in A, that x be in S1 at the start of A and y in S2 at its end, etc. — can be added to the elaborations of appropriate components of the schematic.

Additional referents can be added to a transition link to indicate more information about associated processes. Relative placement on the transition

arrow indicates the relative temporal placement of the associated UOBs. For instance, the schematic in Figure 22 indicates a transition involving the occurrence of a pair of UOBs A and B that start simultaneously, and a third UOB C that starts after A and B. Additionally, because the "B" referent is a call-and-wait, in any instance of the transition, UOB B must complete before C can begin. (This will generally be the only sort of context in which call-and-wait referents are used in transition schematics.)

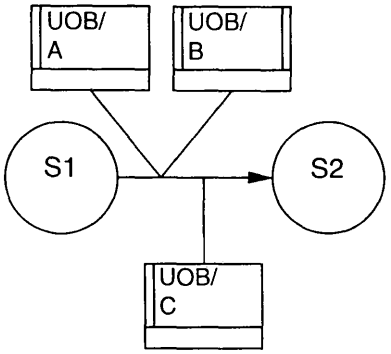

Figure 22: Multiple UOB Referents in a Transition Schematic

The semantics for transitions in schematics with multiple referents is slightly more involved than for simple schematics. In the case of the schematic in Figure 22, for example, the indicated object x in any such transition is in S1 at the start of A and B, and it is in state S2 by the end of C. This semantics generalizes straightforwardly to other cases of multiple referents.

If the relative temporal ordering of the UOBs involved in a transition is unknown or indeterminate from case to case, a small circle is used to "anchor" the referents indicating those UOBs, as illustrated in Figure 23.

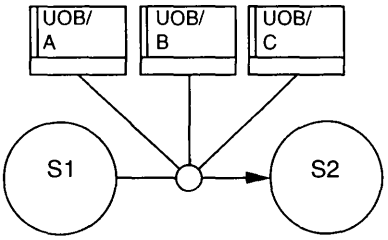

Figure 23: Temporally Indeterminate UOB Referents in a Transition Schematic

It is not uncommon for a given situation to "sustain" an object in a given state; a refrigeration process, for example, might sustain a given substance

in a solid state. Situations of this type can be represented by the construct in Figure 24.

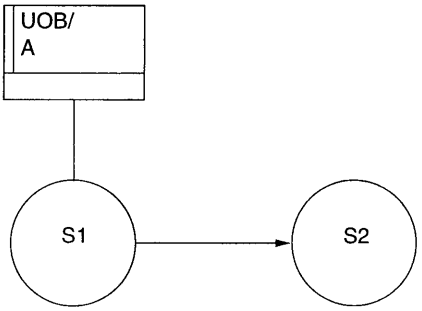

Figure 24: Sustaining an Object in a State

More generally, in any instance of the schematic in Figure 24, there is a UOB A of the sort specified by the referent and an object x in state S1 throughout the duration of A. This requires that such an x must exist when A begins. x could, however, be in state S1 *prior* to the start of A; that is, it could be brought into state S1 by some other process prior to A (the substance noted above might actually become solid through some sort of chemical reaction), and then sustained in that state by A.

5.4.4 Complex Transition Schematics

More complex transition schematics can be constructed by adding further transition arrows and state symbols to existing schematics or by using transition junctions. A complex schematic is illustrated in Figure 25.

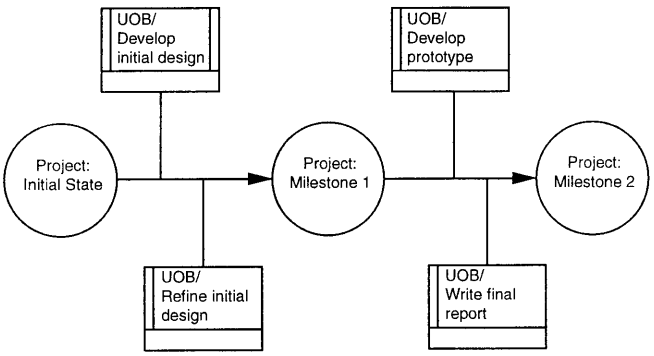

Figure 25: A Complex State Transition Schematic

For the most part, the semantics of complex schematics such as this is a straightforward generalization of simple schematics, only instead of a single transition there are several successive transitions. Thus, the schematic in Figure 25 expresses a transition in which a project evolves from an initial state to a first milestone state and thence to a second milestone state via the UOBs of the sort indicated.

Transition junctions permit the construction of more subtle schematics that express concurrent and alternative paths in a series of transitions. Junctions can take any of the three forms illustrated in Figure 26.

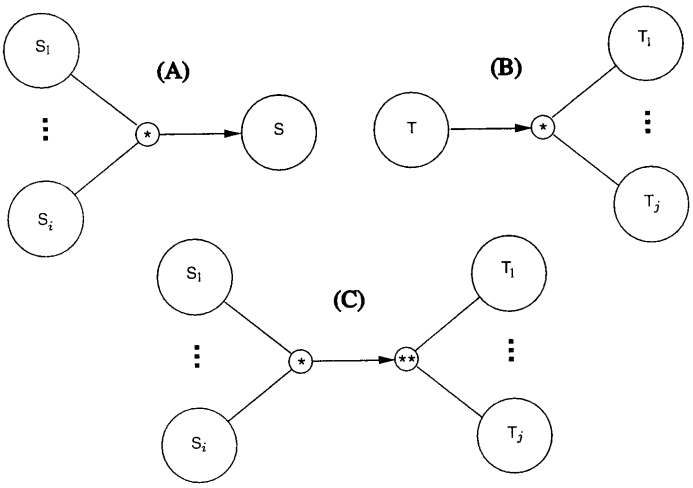

Figure 26: Transition Junctions

The semantics of these junctions parallels their process schematic counterparts. If * is '&' in schematic (A) in Figure 26, for example, then the schematic indicates a transition in which objects $x_1, ..., x_i$ in states $S_1, ..., S_i$, respectively, transition to an object y in state S. If * is 'X' in (B), then the schematic indicates a transition of an object x to an object y in exactly one of the states $T_1, ..., T_j$. Form (C) allows for even more complex transitions. For example, if * is 'O' and ** is '&', then the schematic indicates a transition in which one or more objects $x_1, ..., x_i$ in states $S_1, ..., S_i$ transition to objects $y_1, ..., y_j$ in the states $T_1, ..., T_j$, respectively. Similarly for the remaining possibilities. The syntax and semantics of referents with transition junctions is straightforward but subject to a number of conventions. The reader is referred to [MMP93] for details.

5.5 General Kind Schematics

Early in its development, IDEF3 was focused entirely on the representation of process knowledge, and its language included no transition schematics (see, e.g., [MME94]). The desire of modelers to describe processes from an object-centered perspective led to the introduction of transition schematics. Realization of the importance of general ontologies for understanding, sharing, and reusing process models, however, has led to a deeper integration of the IDEF3 method with the IDEF5 ontology capture method. Indeed, the IDEF5 ontology description language has become incorporated into the IDEF3 transition schematic language. This language permits a modeler to express, not only information about state transitions, but general information about the objects, classes, and relations. Space limitations prevent a detailed discussion of this component of IDEF3. Once again, interested readers are referred to [MMP93].

Acknowledgments: Christopher Menzel would like to thank Alexander Bocast for numerous illuminating discussions concerning IDEF0, and for allowing the authors to borrow heavily from several figures that he designed.

References

[End72] Enderton, H., A Mathematical Introduction to Logic, New York, Academic Press, 1972

[Gru93] Gruber, T., A Translation Approach to Portable Ontologies, Knowledge Acquisition 2, 1993, 199-220

[IEEE97] Standard Users Manual for the ICAM Function Modeling Method — IDEF0, IEEE draft standard, P1320.1.1, 1997

[MMP93] Mayer, R. J., Menzel, C., Painter, M., deWitte, P., Blinn, T., Benjamin, P., IDEF3 Process Description Capture Method Report, Wright-Patterson AFB, Ohio, AL/HRGA, 1993

[MKB95] Mayer, R., Keen, A., Browne, D., Harrington, S., Marshall, C., Painter, M., Schafrik, F., Huang, J., Wells, M., Hisesh, H., IDEF4 Object-oriented Design Method Report, Wright-Patterson AFB, Ohio, AL/HRGA, 1995

[MBM94] Mayer, R., Benjamin, P., Menzel, C., Fillion, F., deWitte, P., Futrell, M., and Lingineni, M., IDEF5 Ontology Capture Method Report, Wright-Patterson AFB, Ohio, AL/HRGA, 1994

[MME94] Menzel, C., Mayer R., Edwards, D., IDEF3 Process Descriptions and

Their Semantics, in: A. Kusiak, C. Dagli (eds.), Intelligent Systems in Design and Manufacturing, New York, ASME Press, 1994

[NIST93a] Integration Definition for Function Modeling (IDEF0), Federal Information Processing Standards Publication 183, Computer Systems Laboratory, National Institute of Standards and Technology, 1993[17]

[NIST93b] Integration Definition for Information Modeling (IDEF1X), Federal Information Processing Standards Publication 184, Computer Systems Laboratory, National Institute of Standards and Technology, 1993

[RB87] Ramey, T., Brown, R., Entity, Link, Key Attribute Semantic Information Modeling: The ELKA Method, ms, Hughes Aircraft, 1987

[Ros77] Ross, D., Structured Analysis (SA): A Language for Communicating Ideas, TSE 3 (1), 1977, 16-34

[Sof81] SofTech, Inc. Integrated computer-aided manufacturing (ICAM) architecture, Pt. II, Vol. V: Information modeling manual (IDEF1), DTIC-B062457, 1981

[17] At the time of this writing, the IDEF0, IDEF1X, IDEF3, IDEF4, and IDEF5 reports listed here are available on the World Wide Web at http://www.idef.com.