IDEF3 Formalization Report

Christopher Menzel
Richard J. Mayer
Douglas D. Edwards

Knowledge Based Systems Laboratory
Texas A&M University

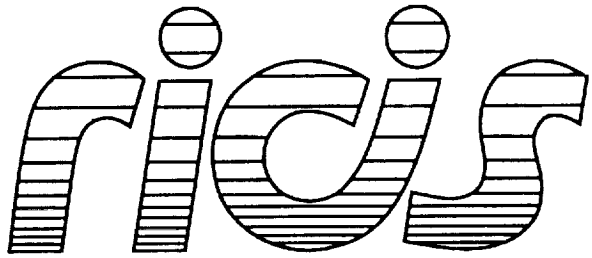
P.57

March 6, 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.

IDEF3 Formalization Report

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Dr. Christopher P. Menzel, Dr. Richard J. Mayer and Dr. Douglas D. Edwards of Texas A&M 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.

IDEF3 Formalization Report

Christopher Menzel, Richard J. Mayer, Douglas D. Edwards
Knowledge Based Systems Laboratory
Texas A&M University

March 6, 1991

Preface

This paper describes the research accomplished 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 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. The authors and the design team wish to acknowledge the technical insights and ideas provided by Captain Painter in the performance of this research as well as his assistance in the preparation of this document.

Summary

The Process Description Capture Method (IDEF3) is one of several ICAM (Integrated Computer-Aided Manufacturing) DEFinition methods developed by the Air Force to support systems engineering activities, and in particular, to support information systems development. These methods have evolved as a distillation of "good practice" experience by information system developers and are designed to raise the performance level of the novice practitioner to one comparable with that of an expert. IDEF3 is meant to serve as a knowledge acquisition and requirements definition tool that structures the user's understanding of how a given process, event, or system works around process descriptions. A special purpose graphical language accompanying the method serves to highlight temporal precedence and causality relationships relative to the process or event being described.

The purpose of the paper is to present a rigorous formalization of the IDEF3 method. Formalization of a method is is accomplished to:

1. Identify the informal intuitions that motivate the method.
2. Provide a technical basis for integration with other methods.
3. Provide an objective basis for the comparison of methods.
4. Provide a technical basis and an engineering technique for the design of new methods.
5. Provide accurate specifications for the design of automated method support tools.

Included within this formalization are definitions of the basic concepts employed by the IDEF3 method and the logic for how these concepts work together to represent a particular view of reality. Also included is a description of the symbols and presentation rules for the graphical component of the method. Together, these provide both a formal and informal description of the Process Description Capture Method and the basic guidelines needed for practical application.

Contents

Introduction

The central way of describing what happens in the world around us, and in particular for describing how a given or prospective system works, is to relate a story in the form of an ordered sequence of events or activities. We call this *process description.* This report is concerned with motivating and formalizing IDEF3, a rigorous method for capturing process descriptions. In particular, IDEF3 is designed (i) to be used by engineering and manufacturing domain experts to express the normal content of a common sense process description, and (ii) to be structured enough to allow for computerized representation, automated interpretation, and intelligent support for uses of the language in capturing process descriptions. The chief motivation for such a method is that, before one can provide any sort of information based application for a user within a particular domain (manufacturing and engineering in particular), one must have an accurate description of the user's understanding of the structure of the domain.

Section 1 of the paper sets the stage for the reporting of our current results. In Section 1.1 we characterize the difference between a model and a description of a part of the world. In Section 1.2 we present some of the phenomena present in typical process descriptions, and briefly discuss the notion of process we will be working with. In Section 1.3 we briefly compare IDEF3 with other process formalisms. Following these three sections we develop IDEF3 formally; that is, we develop IDEF3 as a formal theory, with a syntax and corresponding semantics designed to capture what we consider to be the essential components of process description.

1 Background and Motivation

1.1 Descriptions vs. Models

It is important first to distinguish between models and descriptions (Mayer, 1988). We emphasize that, although models may well be constructed from descriptions, our task here is not the construction of models but the formal representation of descriptions and the information they convey.

To get at the distinction, a model can be characterized as an idealized system of objects, properties, and relations that is designed to imitate in

certain relevant respects the character of a given real world system. The power of a model comes from its ability to simplify the real world system it represents, and to predict certain facts about that system in virtue of corresponding facts within the model.[1] A model is thus in a certain sense a complete system. For in order to *be* an acceptable model of a given or imagined real world situation, it must satisfy certain "axioms" or conditions derived from the real world system.

A description on the other hand is a recording of facts or beliefs about the world around us. As such descriptions are in general partial; a person giving a description may omit facts that do not strike her as relevant, or which she has forgotten in the course of describing the system, etc. There are thus no preconditions on an acceptable description, no "axioms" to be satisfied, short of simple accuracy as far as it goes; descriptions, we might say, are assumed to be true, but incomplete.[2] The accumulation of descriptions is thus prior to and distinct from the construction of models. Indeed, generally, the conditions one puts on acceptable models are derived from descriptions one receives from domain experts; they are, so to say, the data from which models are built.

Descriptions are thus essential to the model building process. An accurate treatment of such descriptions requires two components, one having to do with the descriptions as linguistic entities (the *syntactic* component) and the other having to do with their content, with the information they convey (the *semantic* component). There muat be an effective means of representing the descriptions themselves, a means of capturing their "logical form," and a rigorous account of their information content.

On the syntactic side, as we will see, IDEF3 makes use of the standard language of first order logic as its formal base. This permits a rich and flexible means of expressing the logical form of most any typical descriptive statement. Semantically, we use a variant of first-order semantics, enriched to represent the temporal information so crucial to process descriptions. This approach, unlike that of typical simulation languages, enables us to interpret the intended meaning of a given descriptions in terms of a semantic structure that corresponds in a natural way to the real world situation being described.

[1] See (Corynen, 1975) for a detailed analysis of this phenomena.

[2] In fact one very powerful use of models is to fill in the gaps in our descriptions.

1.2 Phenomenological and Linguistic Motivations

1.2.1 What is a Process?

To understand the idea of capturing process descriptions, one must first know what we mean by a process. We are not using the term in a technical sense, but in an ordinary-language sense, in keeping with IDEF3's role as a methodology for acquiring the intuitive knowledge of domain experts. Unfortunately, the term "process" is quite ambiguous in English. We will need to refine our understanding of process terminology in ordinary language before we can characterize the intended sense of "process" more clearly.

Since we are interested in capturing a human's understanding of the the world around him (and how it works) it is necessary to characterize the concept of a process in view of that understanding. Such a characterization is bound to be difficult since the notion of *change*—a notoriously slippery notion—is basic to the concept of process. Intuitively, the term process is used to describe an isolable event or occurrence. As such it can be assigned a more or less definite starting point (typically associated with the satisfaction of certain antecedent conditions) and continue indefinitely. A process will in general involve objects with certain (perhaps changing) properties standing in specified (perhaps changing) relations. A process can also stand in relations with other processes: e.g., a process can start, suspend and terminate other processes; objects or information about objects can be shared between processes; one process can change the properties of such a shared object and "cause" the exclusion of another process execution; etc.

It is crucially important to distinguish between *process types* and *process instances*, or *individual processes*. (Indeed, as we'll see below, it is important to distinguish generally between types and instances with regard to many other kinds of entities as well.) We think of an *individual* process as a concrete occurrence located at a specific time and place. Process *types* may be thought of as *classes* of individual processes or *properties* that individual processes may have. It is unfortunate that the English language does not distinguish between process types and individual processes; the word "process" can refer to either. In this essay, however, we will attempt to maintain the distinction rigorously whenever it matters. (It does not always matter; for instance, it does not matter in very general contexts (as above) or where the term "process" occurs as an integral part of phrases like "process flow

description capture" or "process model.") Process types may vary from general to specific. An individual process p which is among those picked out by a process type P will be called an *instance* of p. If process types P and Q are such that any instance of P is also an instance of Q, then P is said to be a *subtype* of Q. Similar terminology is used for types and individuals of other varieties than processes.

One important note about the more general use of the term *individual*: not all individuals are concrete entities. Some types, like *number*, may have instances which are abstract entities; these instances are nonetheless individuals. In fact, the term "individual" is really more or less synonymous with "instance"; a thing is called an individual only with a tacit reference to some type of which it is an instance.

1.2.2 Narrow and Broad Senses of "Process"

Another problem in describing processes in English arises from the language's intense focus on the details of temporal succession, characteristic of the Indo-European languages.

There is one sense of "process" in which the word is distinguished from "event," "state of affairs," "eventuality," "occurrence," and any number of other words of this general class. In fact, there are several such senses of "process," each stressing a different kind of distinction between processes and other things of this general kind. For instance, in one sense processes are supposed to have internal structure, as opposed to events, which are pointlike. In another sense, each instance of a given process type are supposed to be divisible into temporal subparts which are process instances of the same process type, whereas this would not be true for events. The linguist and philosopher Zeno Vendler (Vendler, 1967; Vendler, 1968), among others, has gone into great detail in classifying things of this kind, coining individual terms for concepts represented by variations in meaning in English.

On the other hand, there is another sense of "process" in which it is synonymous with "event." In this sense the extensions of any of the other terms listed in the preceding paragraph would be subsets of the class of processes; a process is *anything* of the general kind described in the preceding paragraph. It is this broadest sense of 'process' with which we are concerned in IDEF3; the Vendlerian classification is irrelevant to our purposes. (IDEF3 has its own ways of distinguishing one kind of process from another based on

the internal structure of the instances.) The precise technical term we have invented for processes (in this broad sense) is *unit of behavior* (UOB), which simply means *a process or event, in the most general senses of those terms.* We continue to write simply "process" where we feel it will be clear that any UOB is meant, not just a process in some narrower sense.

1.3 IDEF3 vs. Other Process Formalisms

Various disciplines have their own special perspectives on the task of describing processes. In the world of simulation, for instance, "modeling" a process means constructing a model which can be used to simulate the process. In the literature on robot planning in artificial intelligence, a plan is a kind of process description.

Process modeling and planning, however, are only two ways in which processes might be described. IDEF3 aims at producing high-level, general-purpose descriptions of processes. These are *IDEF3 models*, not to be confused with "process models" in the simulation sense. There are many purposes for which IDEF3's approach is useful including:

- Determination of the impact of an organization's information resources on the major operation scenarios of the business.
- Documentation of the decision procedures affecting the states and life cycles of critical shared data.
- Organization of the user supplied descriptions of user operations to assist in requirements decision making and system design.

Additionally, an IDEF3 model can be used in the early stages of defining a knowledge based system, a simulation study, a robot plan, or some other kind of special-purpose model of a process. This use of IDEF3 as an *early, fact-gathering and organization aid* can save time and reduce complexity in early design. As in every design activity, it is easier to discover *how* to do something once you know *what* you want to do. Furthermore, because IDEF3 carries less of the technical baggage of various entrenched disciplines than do most other process formalisms, it will be relatively easy for a domain expert, without extensive training in any process formalism, to use IDEF3 to *communicate* with designers of many different kinds of systems

(software, simulation, shop floor machining systems, etc.). IDEF3 will thus be a powerful tool for knowledge acquisition.

With this background, then, we now move on to a more precise account of IDEF3. We begin with some formal prerequisites.

2 The Syntax and Semantics of First-order Logic

IDEF3 diagrams have a definite syntax. Furthermore, diagrams constructed in accordance with that syntax are intended to represent certain chunks of the world accurately and informatively; the diagrams, that is, have semantic content. All too often the syntax of information representations of various kinds is ill-defined, and the intended semantic content of such representations (how they are supposed to hook up with the world) is vague and imprecise. We attempt to avoid these problem in IDEF3 by providing an explicitly defined syntax to make clear exactly what does and what does not count as an IDEF3 diagram and corresponding mathematically precise semantics to make clear exactly what sorts of structures are representable by IDEF3 diagrams.

To help us achieve this goal, we rely heavily upon the syntax and semantics of first-order logic. There are several reasons for this decision. First, first-order logic is as clearly understood as any extant scientific or mathematical theory. This enables us to proceed in confidence that the most basic theoretical foundations of our work are sound. Second, although the language of first-order logic was originally designed to express propositions of mathematics with clarity and precision, it soon became clear that much of natural language could also be clearly represented in this formal language. This is especially true for constrained fragments of natural language such as one might find in a manufacturing, engineering, or database setting,[3] and hence makes first-order logic a natural and effective choice for capturing the propositional content of process descriptions in a rigorous and precise way. Finally, for all its rigor, the theory of first-order logic is intuitive and rela-

[3]There is in particular a very large literature on the uses of logic in database design. See, for example, (Frost, 1986), ch. 5, for a good introduction to logical database theory, and (Gallaire and Minker, 1978) or (Jacobs, 1982) for more detailed treatments.

tively simple to understand. The language is a straightforward idealization of ordinary discourse, and its semantics, or *model theory*, provides especially natural mathematical representations of the phenomena to be modelled.

Since we are assuming no familiarity with first-order logic, this section of our paper will cover enough of the basic theory to enable one to follow the exposition of the foundations of IDEF3 to follow. Some knowledge of basic set theory will be presupposed.

2.1 First-order Languages

First-order logic is expressed in a *first-order language*. Such a language $\mathcal{L}$ is a *formal* language. That is, it is a formal object consisting of a fixed set of basic symbols, often called the *vocabulary* of $\mathcal{L}$, and a precise set of syntactic rules, its *grammar*, for building up the sentences, or *formulas*, of the language, those syntactic objects that are capable of bearing information.

2.1.1 Vocabulary

The basic vocabulary of a first order language consists of several kinds of symbols:[4]

- Constants
- Variables
- Predicates
- Logical symbols.

Constants are symbols that correspond to names in ordinary language. For many purposes, it is useful to use abbreviations of names straight out of ordinary language for constants, e.g., j for John, wp for Wright-Patterson, v for Venus, e for Elevator 1, etc. When we are describing languages in general and have no specific application in mind, we will simply use the letters a, b, c, and d, perhaps with subscripts; we will assume that we will add no

[4]We omit function symbols for purposes here, though they would be present in a complete account of the theory.

more than finitely many subscripted constants to our language.[5] Constants are usually lower case letters, with or without subscripts, but this is not necessary. Indeed, it is often useful to use upper case.

We will often want to say things about an "arbitrary" constant as a way of talking about all constants, much as one might talk about an arbitrary triangle ABC in geometry as a way of proving something about all triangles in general. For this purpose it will not do to talk specifically about a given constant, a say, since we want what we say to apply to all constants. This requires that, when we are talking *about* our language, that we use special *metavariables* whose roles are to serve as placeholders for arbitrary constants of our language, much as ABC above serves as a placeholder for arbitrary triangles. Thus, metavariables are not themselves part of our first-order language $\mathcal{L}$, but rather part of the extended English we are using to talk about the constants that *are* in the language. We will use lower case *sans serif* characters a, b, c for this purpose.

Next on the list are the variables, whose purpose will be clarified in detail below. The lower case letters x, y, and z, possibly with subscripts, will play this role, and we will suppose there to be an unlimited store of them. We will use the characters x, y and z as metavariables over the store of variables in our language.

The third group of symbols in our language consists of n-place predicates, $n > 1$. One-place predicates correspond roughly to verb phrases like "has insomnia," "is an employee," "is activated," and so forth, all of which express properties. Two-place predicates correspond roughly to transitive verbs like "begat," "is an element of," "weighs less than," "enters," and "lifts," and these express two-place *relations* between things. There are also three-place relations, such as those expressed by "gives" and "between," and with a little work we could come up with relations of more than three places, but in practice we have little cause to go much beyond this.

When speaking generally, for predicates we will use upper case letters such as P, Q, and R. Occasionally these may appear with numerical superscripts to indicate the number of places of the relation they represent, and if necessary with subscripts to distinguish between those with the same superscripts.

[5] The restriction to a finite number of constants here is not at all essential, but constraint languages in general will use only finitely many; the same holds for predicates and function names below.

Once again, though, in practice it is often useful to abbreviate relevant natural language expressions. Most languages contain a distinguished predicate for the two-place relation "is identical to." We will use the symbol $\approx$ for this purpose. Once again, the corresponding *sans serif* characters P, Q, R etc. will serve as metavariables.

The last group of symbols consists of the basic logical symbols: $\neg$, $\wedge$, $\vee$, $\supset$, and $\equiv$, about which we shall have more to say shortly. (For ease of exposition here we will omit quantifiers, though these would be present in a thorough treatment.) We will also need parentheses and perhaps other grouping indicators to prevent ambiguity.

2.1.2 Grammar

Now that we have our basic symbols, we need to know how to combine them into grammatical expressions, or *well-formed formulas*, which are the formal correlates of sentences. These will be the expressions that will encode the propositional content of process descriptions in our theory (and more). This is done recursively as follows.[6]

First, we want to group all name-like objects into a single category known as *terms*. This group includes the constants of course, and for reasons discussed below, it includes the variables as well.

Next we define the basic formulas of our language. Just as verb phrases and transitive verbs in ordinary language combine with names to form sentences, so in our formal language predicates combine with terms to form formulas. Specifically, if P is any n-place predicate, and $\mathsf{t}_1, \ldots, \mathsf{t}_n$ are any n terms, then $\mathsf{P}\mathsf{t}_1 \ldots \mathsf{t}_n$ is a formula, and in particular an *atomic* formula. To illustrate this, if H abbreviates the verb phrase "is happy," and a the name "Annie," then the formula Ha expresses the proposition that Annie is happy. Again, if L abbreviates the verb "loves," b the name "Barbara," c the name "Charlie," then the formula Lbc expresses the proposition that Barbara loves Charlie. We will use the lower case Greek letters ϕ, ψ, and θ as metavariables over formulas.

Often when one is using more elaborate predicates drawn from natural language, e.g., if we had used *LIFTS* instead of L in the previous example, it is more readable to use parentheses around the terms in atomic formulas that

[6] We will say a little more about recursive definitions below.

use the predicate and separate them by commas, e.g., $LIFTS(b,x)$ instead of $LIFTSbx$. Thus, more generally, any atomic formula $\mathsf{Pt}_1 \ldots \mathsf{t}_n$ can be written also as $\mathsf{P}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$. Furthermore, atomic formulas involving some familiar two-place predicates like $\approx$, and a few others that will be introduced below, are more often written using *infix* rather than *prefix* notation, i.e., with the predicate between the two terms rather than to the left of them. For example, we usually express that a is identical to b by writing $a \approx b$ rather than $\approx ab$. Thus, we stipulate that formulas of the form $\mathsf{Ptt'}$ can also be written as $\mathsf{tPt'}$.

Now we begin introducing the logical symbols that allow us to build up more complex formulas. The symbol $\neg$ expresses negation; i.e., it stands for the phrase 'It is not the case that'. Since we can negate any declarative sentence by attaching this phrase onto the front of it, we have the corresponding rule in our formal grammar that if φ is any formula, then so is $\neg\varphi$. The symbols $\wedge$, $\vee$, $\supset$, and $\equiv$ stand roughly for "and," "or," "if...then," and "if and only if," which are also (among other things) operators that form new sentences out of old in the obvious ways. Unlike negation, though, each takes *two* sentences and forms a new sentence from them. Thus, we have the corresponding rule that if φ and ψ are any two formulas of our language, then so are $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \supset \psi)$, and $(\varphi \equiv \psi)$. So, to illustrate once again, using the abbreviations above, $(Lcb \supset (Hb \wedge \neg Ha))$ expresses that if Charlie loves Barbara, then Barbara is happy and Annie is not.

Finally, we turn to the quantifiers $\exists$ and $\forall$. Recall that we introduced variables without explanation above. $\exists$ and $\forall$ stand for "some" and "every," respectively; one central task of the variables is to enable them to play this role in our formal language. (They shall have another, crucial role to play in IDEF3, as we will see.) Consider the difference between "Annie is happy," "Some individual is happy," and "Every individual is happy." In the first case, a specific individual is picked out by the name "Annie" and the property of being happy is predicated of her. In the second, all that is stated is that some unspecified individual or other has this property. And in the third, it is stated that every individual, whether specifiable or not, has this property. This lack of specificity in the latter two cases can be made explicit by rephrasing them like this: for some (resp., every) individual x, x is happy. Since the rule for building atomic formulas counted variables among the terms, we have the means for representing these paraphrases. Let H abbreviate "is happy" once again; then we can represent the paraphrases

as $\exists x Hx$ and $\forall x Hx$ respectively.

Accordingly, we add the final rule to our grammar: if φ is any formula of our language and x is any variable, then $\exists x\varphi$ and $\forall x\varphi$ are formulas as well. In such a case we say that the variable x is *bound* by the quantifier $\exists$ (resp., $\forall$), and we say that the formula φ is the *scope* of the quantifer $\forall$ in $\forall x\varphi$, and it is the scope of the quantifier $\exists$ in $\exists x\varphi$.

2.2 First-order Semantics: Structures

We have motivated the construction of our grammar by referring to the intended meanings of the logical symbols and by letting our constants and variables abbreviate meaningful expressions out of ordinary language. But from a purely formal point of view, all we have in a language is uninterpreted syntax; we have not described in any formal way how to assign meaning to the elements of a first-order language. We will do so now.

A *structure* for a first-order language $\mathcal{L}$ consists simply of two elements: a set $\mathcal{D}$ called the *domain* of the structure, and a function V known as an *interpretation function* for $\mathcal{L}$. $\mathcal{D}$ is the set of things one is describing with the resources of $\mathcal{L}$, e.g., the natural numbers, major league baseball teams, objects flowing through a manufacturing system, the people and objects that make up an air force base, or the records inside a database. The purpose of V is to fix the meanings of the basic elements of $\mathcal{L}$—in our present case, constants and predicates—in terms of objects in or constructed from $\mathcal{D}$.

2.2.1 Interpretations for Constants and Predicates

Constants Variables will not receive a specific interpretation, since their meanings can vary within a structure (they are *variables* after all). They will be treated with their own special, but related, semantic apparatus below. Constants, being the formal analogues of names with fixed meanings, are assigned members of $\mathcal{D}$ once and for all as their interpretation; in symbols, for all constants c of $\mathcal{L}$, $V(\mathsf{c}) \in \mathcal{D}$.

Predicates For any one-place predicate P, we let $V(\mathsf{P})$ be a subset of $\mathcal{D}$—the set of things that have the property expressed by P. And for any n-place predicate R, $n > 1$, we let $V(\mathsf{R})$ be a set of n-tuples of elements of $\mathcal{D}$—the set of n-tuples of objects in $\mathcal{D}$ that stand in the relation expressed by

R. For example, if we want 'L' to abbreviate the verb "loves," then if our domain $\mathcal{D}$ consists of the population of Texas, then $V(L)$ will be the set of all pairs $\langle \mathbf{a}, \mathbf{b} \rangle$ such that **a** loves **b**. Formally, then, for all n-place predicates P, $V(\mathsf{P}) \subseteq \mathcal{D}^n$, where $\mathcal{D}^n$ is the set of all n-tuples of elements of $\mathcal{D}$.

If one wishes to include the identity predicate $\approx$ in one's language, and have it carry its intended meaning, then one needs an additional, more specific semantical constraint on the interpretation function V. Identity, of course, is a relation that holds between any object and itself, but not between itself and any other object. This additional constraint is easy to express formally: if our language $\mathcal{L}$ contains $\approx$, then the interpretation of $\approx$ is the set of all pairs $\langle o, o \rangle$ such that o is an element of the domain $\mathcal{D}$, i.e., more formally, $V(\approx) = \{\langle o, o \rangle \mid o \in \mathcal{D}\}$.

2.2.2 Truth under an Assignment

Our goal now is to define what it is for a formula to be *true* in a given a structure $\mathrm{M} = \langle \mathcal{D}, V \rangle$. To do so, we will first need the notion of a *variable assignment*, or *assignment* for short. An assignment might be thought of as a "temporary" interpretation function for variables: like an interpretation function on constants, it assigns members of the domain to variables; but within the same structure we make use of many different assignment functions. This reflects the semantic variability of variables as opposed to constants and predicates. Now, given our structure M and an assignment α, we can define interpretations for *terms*, i.e., constants *and* variables generally, relative to α: the interpretation $V_\alpha(\mathsf{t})$ of a term t under an assignment α is just $V(\mathsf{t})$, if t is a term, and $\alpha(\mathsf{t})$—the object in $\mathcal{D}$ assigned to t by α—if t is a variable.

Atomic Formulas Given a general notion of an interpretation for terms under an assignment α, we can now define the notion of *truth under an assignment* in a structure M. Truth *simpliciter* in M will then be defined in terms of this notion. For convenience, we will often speak of a formula's being "true$_\alpha$ in M" instead of being "true in M under α."

We start by defining truth under an assignment for atomic formulas. So let φ be an atomic formula $\mathsf{Pt}_1 \ldots \mathsf{t}_n$. Then φ is true$_\alpha$ in M just in case $\langle V_\alpha(\mathsf{t}_1), \ldots, \mathcal{F}_\alpha(\mathsf{t}_n) \rangle \in V_\alpha(\mathsf{P})$. Intuitively, then, where $n = 1$, Pt is true$_\alpha$ in M just in case the object in $\mathcal{D}$ that t denotes is in the set of things that have

the property expressed by P. And for $n > 1$, $\mathsf{Pt}_1 \ldots \mathsf{t}_n$ is true_α just in case the n-tuple of objects $\langle o_1, \ldots, o_n \rangle$ denoted by $\mathsf{t}_1, \ldots, \mathsf{t}_n$ respectively is in the set of n-tuples whose members stand in the relation expressed by P, i.e., just in case those objects stand in that relation.

To help fix these ideas, let us actually construct a small language $\mathcal{L}^*$ and build a small structure M*. Suppose we have four names a, b, c, d, a one-place predicate H (intuitively, to abbreviate "is happy"), and a three-place predicate T (intuitively, to abbreviate "is talking to ... about"). Let us also include the distinguished predicate $\approx$, though we will make no real use of it until later. We will use x, y, and z for our variables.

For our structure M*, we will take our domain $\mathcal{D}$ to be a set of three individuals, {Beth, Charlie, Di}, and our interpretation function $\mathcal{G}$ will be defined as follows. For our constants, $\mathcal{G}(a) = \mathcal{G}(b) =$ Beth, $\mathcal{G}(c) =$ Charlie, and $\mathcal{G}(d) =$ Di. (Beth thus has two names in our language; this is to illustrate a point to be made several sections hence.) For our predicates H and T, we let $\mathcal{G}(H) =$ {Beth, Di} (so, intuitively, Beth and Di are happy), and $\mathcal{G}(T) =$ {⟨Beth, Di, Charlie⟩, ⟨Charlie, Charlie, Di⟩} (so, intuitively, Beth is talking to Di about Charlie, and Charlie is talking to himself about Di). Following the rule for $\approx$, we let $\mathcal{G}(\approx) =$ {⟨Beth, Beth⟩, ⟨Charlie, Charlie⟩, ⟨Di, Di⟩}. Finally, for our assignment function α, we let $\alpha(x) = \alpha(y) =$ Charlie, and $\alpha(z) =$ Di.

Let us now check that Hd and $Tbdx$ are true in M* under α. In the first case, by the above, Hd is true_α in M* just in case $\mathcal{G}_\alpha(d) \in \mathcal{G}_\alpha(H)$, i.e., just in case Di is an element of the set {Beth, Di}, which she is. So Hd is true_α in M*. Similarly, $Tbdx$ is true_α in M* just in case $\langle \mathcal{G}_\alpha(b), \mathcal{G}_\alpha(d), \mathcal{G}_\alpha(x) \rangle \in \mathcal{G}_\alpha(T)$, i.e., just in case $\langle \mathcal{G}(b), \mathcal{G}(d), \alpha(x) \rangle \in \mathcal{G}(T)$, i.e., just in case ⟨Beth, Di, Charlie⟩ ∈ {⟨Beth, Di, Charlie⟩, ⟨Charlie, Charlie, Di⟩}. Since this obviously holds, the formula $Tbdx$ is true_α in M*.

A formula is $\textit{false}_\alpha$ in a structure M, of course, just in case it is not true_α in M. It is easy to verify that, for example, Hc, Hx, and $Tdbc$ are all false_α in M* under α.

Digression on Variables, Types, and Instances We emphasized above the disinction between types and instances. It is important to see how this distinction is captured to a certain extent by the apparatus of variables and assignments. Though we have no specific semantic object corresponding to

types, *intuitively* formulas with unassigned variables can be thought of as expressing types of situations.[7] For example, the formula Hx, *independent of any assignment*, can be thought of as expressing the type of situation in which someone or other is happy; when that variable is assigned, or *anchored*, to a given individual, Charlie say, then we get a determinate instance of that situation type. Again, $Tzcw$ can be thought to express the type of situation in which someone is talking to Charlie about someone, an assignment of some object, Beth say, to z yields the somewhat more determinate situation type in which Beth is talking to Charlie about someone or other, and a further assignment of an object to w then yields a determinate instance of the latter two types.

This reflection of the type/instance distinction in the assigned/unassigned variable distinction plays a crucial role in representing the process-type/process-instance distinction in IDEF3. We will have more to say about this presently.

Conjunctions, Negations, etc. Now for the more complex cases. Suppose first that φ is a formula of the form $\neg\psi$. Then φ is true_α in a structure M just in case ψ is *not* true_α in M. In so defining truth for negated formulas we ensure that the symbol $\neg$ means what we have intended. Things are much the same for the other symbols. Thus, suppose φ is a formula of the form $\psi \wedge \theta$. Then φ is true_α in M just in case both ψ *and* θ are. If φ is a formula of the form $\psi \vee \theta$, then φ is true_α in M just in case either ψ *or* θ is. If φ is a formula of the form $\psi \supset \theta$, then φ is true_α in M just in case either ψ is false in M or θ is true_α in M. And if φ is a formula of the form $\psi \equiv \theta$, then φ is true_α in M just in case ψ and θ have the same truth value in M.

The reader should test his or her comprehension of these rules by verifying that $\neg H(y)$ and $(Tbcz \wedge Txxc) \supset Hc$ are both true in $\mathbf{M}^*$ under α.

Quantified Formulas Last, we turn to quantified formulas. (This section can be omitted without impairing the reader's understanding, since our examples below do not involve quantified statements.) The intuitive idea is this. When we introduced the quantifiers above, we noted that "Some in-

[7] The notion of a situation is at the heart of much recent work in natural language semantics, philosophy, logic, and artifical intelligence, especially around Stanford University. See esp. (Barwise and Perry, 1983) and (Barwise, 1989) in this regard.

dividual is happy," i.e., $\exists x Hx$, can be paraphrased as "for some individual x, x is happy." This in turn might be parapharased more linguistically as "for some value of the variable x, the expression 'x is happy' is true." This is essentially what our formal semantics for existentially quantified formulas will come to. That is, $\exists x Hx$ will be true in a structure M under α just in case the *un*quantified formula Hx is true in M under some (in general, new) assignment α' such that $\alpha'(x)$ is in the interpretation of H. It is easy to verify that this formula is true in our little structure M* under α, when we look at a new assignment function α' that assigns either Beth or Di to the variable x. Thus, $\exists x Hx$ should come out true in M* under α.

But we have to be a little more careful in defining truth in a structure formally, because some formulas—$Tbxz$, for example—contain more than one unquantified, or *free*, variable. Thus, when we are evaluating a quantification of such a formula—$\exists z Tbxz$, say—we have to be sure that the new assignment function α' doesn't change the value of any of the free variables—in this case, the variable x. Otherwise we could change the sense of the unquantified formula in mid-evaluation. So, intuitively, under the assignment α above, $\exists z Tbxz$ intuitively says that Beth is talking to Charlie about someone (recall that $\alpha(x)$ = Charlie), and this should turn out to be false_α in M* since Beth is not talking to Charlie about anyone, i.e., there is no triple in $\mathcal{F}(T)$ such that Beth is the first element and Charlie the second. But suppose all we require to make an existentially quantified formula true under α is that there be some new assignment function α' such that $Tbxz$ is true under α'. Then it could turn out also that $\alpha'(x)$ is Di and $\alpha'(z)$ is Charlie. But then the formula $Tbxz$ would be true in M* under α', since Beth is talking to Di about Charlie, i.e., (Beth, Di, Charlie) $\in \mathcal{G}_\alpha(T)$. And that is clearly not what we want.

All that is needed to avoid this problem is a simple and obvious restriction: when evaluating the formula $\exists z Tbxz$, the new assignment α' that we use to evaluate $Tbxz$ must not be allowed to differ from α on any variable except z (and even then it *needn't* differ from α). More generally: if φ is an existentially quantified formula $\exists x\psi$, then φ is true in a structure M under α just in case there is an assignment function α' just like α except perhaps in what it assigns to x such that the formula ψ is true in M under α'. And if φ is a universally quantified formula $\forall x\psi$, then φ is true in M under α just in case for *every* assignment function α' just like α except perhaps in what it assigns to x the formula ψ is true in M under α'. That is, in essence, φ is

true in M under α just in case ψ is true in M no matter what value in the domain we assign to x (while keeping all other variable assignments fixed).

The reader can once again test his or her comprehension by showing in detail that $\exists x T x b y$ is false$_\alpha$ in M*, and that $\forall x(Hx \vee Tbdx)$ is true$_\alpha$ in M*.

2.2.3 Truth and Realization

Now, finally, we can define a formula to be *true* in a structure M *simpliciter* just in case it is true$_\alpha$ in M for all assignments α, and *false* in M just in case it is false$_\alpha$ in M for all α. Note that for most any interpretation, there will be formulas that are neither true nor false in the interpretation. Our example $\exists z T b x z$ above, for instance, is neither true nor false in M*. Such formulas will of course always have free variables, since it is the semantic indeterminacy of such variables that is responsible for this fact. However, note that some formulas with free variables—e.g., $Hx \wedge \neg Hx$—will nonetheless be true or false in certain models, though these will typically be logical truths (resp., falsehoods), i.e., formulas which are not capable of true (resp., false) interpretation.

A structure M is said to be a *realization* of a given set Σ of formulas just in case every formula in Σ is true in M.[8] So, for example, our structure M* is a realization of the set $\{Hd,\ Hx \supset Hx,\ Tbdc \wedge \neg Hc,\ \forall x(Hx \vee \exists y(Tydx))\}$. The notions of truth and realization will be central to our semantics for IDEF3.

2.3 Temporality and Index Semantics

2.3.1 First-order Logic and the Problem of Temporality

Despite its success in numerous domains, there are areas of potential application where plain vanilla first-order logic and its semantics comes up short. Most notable among these are domains involving time. A standard first-order structure "freezes" the world as it were at a certain moment, and this seriously hinders the representation of dynamic processes. For our purposes,

[8] It is more common in logic to say that such a structure is a *model* of Σ. But this term is already so overextended in the area of information modeling (case in point) that we thought it best to avoid the standard terminology here.

then, first-order logic needs some supplementation. The answer, or at least one good answer, is *index* semantics.

Roughly speaking, index semantics is simply an expanded first-order semantics. That is, instead of a single interpretation, in an index semantical structure one finds many plain interpretations, each distinguished from the other (in addition to internal differences) by a unique index. The idea then, in the temporal case, is that each index can represent a certain moment, or a certain interval, of time, and the structure it indexes represents the world at that moment or during that interval.[9] In effect, one overcomes the static representation of a single first-order interpretation by stringing together a series of related snapshots. The result, though somewhat artificial, nonetheless adds great expressive power and flexibility to unadorned first-order languages and their semantics.

2.3.2 Elaborations

We now introduce extensions to our language and our semantics appropriate to the task at hand. To our language we add a new class of temporal constants $k_1, k_2, \ldots$ and temporal variables $i_1, i_2, \ldots$, and a distinguished class of n-place temporal predicates. The temporal constants will serve as names of intervals, e.g., 12 noon (on a particular day), 9-12 a.m., etc. Temporal variables will of course take temporal intervals as values, and the predicates intuitively express properties of, and relations among, intervals, e.g., duration properties like *five minutes in length*, and significant temporal relations (precedence and inclusion in particular).

Temporal terms and predicates will not be allowed in formulas of our original language. Rather, they are the elements of a separate, and for purposes here, simpler temporal language whose only logical symbols are the boolean connectives (and hence there are no temporal quantifiers). The resulting formulas will be called *temporal* formulas, and formulas from our original language *standard* formulas. Formulas of both sorts will be used

[9]Though there were several precursors to full-blown index semantics, most notably the work of Prior (Prior, 1957), it reached its maturity in the "possible world" semantics that Saul Kripke provided for modal languages, i.e., languages with such operators as "necessarily," "possibly," "It has always been the case that," "It will be the case that," etc. See (Kripke, 1963) for a readable overview, (Chellas, 1980) and (Hughes and Cresswell, 1968) for more formal developments.

in the construction of a new kind of expression that we call an *elaboration*, consisting of a temporal interval variable, temporal formulas involving that variable, and a collection of nontemporal formulas. More specifically, let i be a temporal interval variable, $\psi_1, \ldots, \psi_n$ $(n \geq 0)$[10] temporal formulas, and $\varphi_1, \ldots, \varphi_m$ $(m > 0)$ are standard formulas, $[\mathrm{i}, \{\psi_1, \ldots, \psi_n\}, \{\varphi_1, \ldots, \varphi_m\}]$ is an elaboration.[11] i will be called the *dominant* temporal variable of the elaboration.

To anticipate things a bit, the temporal formulas in an elaboration put conditions on the value of the temporal variable i, e.g., that it be before noon, or have a duration of twenty minutes. Subject to these conditions, the elaboration is to be thought of as "asserting" that each of its standard formulas is true throughout the value of i. An elaboration thus represents (at a certain level of detail) what is occurring within a particular temporally extended situation—type or instance, depending on whether or not there are free variables occurring in the elaboration. An instance of a process, then, thought of roughly as a sequence of actual events, can be represented by a corresponding sequence of "determinate" elaborations—elaborations containing no free variables. A general process *description* will be represented by a structured cluster of "indeterminate" elaborations—elaborations with free variables among the formulas—that can be instantiated in many different ways, corresponding to the different possible runs of the general process captured in the description. The graphical syntax of these clusters will be described below.

2.3.3 Temporal Structures

To be able to represent these ideas semantically, we add temporal indices to our plain structures.. Specifically, a *temporal* structure $\langle D, TI, dom, V \rangle$ consists now not only of a domain D and an interpretation function V, but two other elements as well. First, there is a set of temporal intervals, or more exactly, a triple $TI = \langle T, \leq, \sqsubseteq \rangle$, where T is a set, and $\leq$ and $\sqsubseteq$ are two binary relations on T—intuitively, $\leq$ represents the relation of temporal

[10] So in the case where $n = 0$, the sequence $\psi_1, \ldots, \psi_n$ is the empty sequence.

[11] The restriction to temporal *variables* i is not a genuine restriction, since for any temporal constant k we can include the condition i = k among the formulas ψ_j, thus in effect eliminating the restriction. Using only variables in the definition, however, makes for a smoother statement of the semantics for *instantiation graphs* below.

precedence (last Tuesday precedes last Thursday), and $\sqsubseteq$ the relation of temporal inclusion (today's lunch hour is included in the period from 8:00 this morning to 5:00 this evening). Familiar temporal properties and relations can be defined in terms of these notions. For example, an *atomic* interval can be defined as an interval that includes no intervals but itself.

We will impose certain further conditions on the temporal structure of our intervals. In particular, we will assume that every interval has a beginning (and ending) point. This can be stated in the above terms as the condition that every interval τ includes an interval that precedes (is preceded by) every other interval included in τ. Under this condition we can say that one interval τ *meets* another τ' just in case the ending point of τ is the beginning point of τ'.

In most real world settings, not every object exists across every temporal interval. In life, people die, others are born; in a manufacturing system, new objects are constructed, others leave the system; in a database, new records are added, old ones deleted; and so on. Thus, we want to have the flexibility to have different domains of objects associated with different intervals of time; more exactly, we want the plain structures indexed by different intervals to be able to have different domains. That is the job of the second element *dom* in temporal structures. Specifically, in a temporal structure $\langle D, TI, dom, V\rangle$, D is to be thought of as the set of all the objects that exist in *any* of the temporal intervals represented by TI (that is, all the objects that exist in any temporal interval in the real world setting that the structure is designed to represent). *dom* then assigns to each interval in TI the set of objects that exist in that interval—that is, intutively, the objects that exist (at least) from the beginning of the interval to the end and at every point in betweeen. The same object, of course, might, and generally will, exist in many different temporal intervals. Thus, if we were modelling the run of a manufacturing system over a twenty-four hour period, D would consist of all the objects that occur in the system during any interval within that period—parts, employees on their various shifts, finished jobs from the time of their completion to the time they leave the system, etc.

Just as we place conditions on $\leq$ and $\sqsubseteq$ to ensure that they capture the properties of temporal intervals that we wish to represent, we must also place a similar condition on *dom*. Specifically, if an object exists throughout a certain interval of time, then it exists throughout every subinterval. However, nothing we have said so far about our formal structures proper guarantees

that *dom* will represent this fact, i.e., that it will assign all the objects in a given interval τ also to every subinterval of τ. We guarantee this with the following condition, stated explicitly in terms of our semantical apparatus: for any interval $\tau \in T$, and for any object $o \in D$, if $o \in dom(\tau)$, then for any $\tau' \in T$, if $\tau' \sqsubseteq \tau$, then $o \in dom(\tau')$. We don't require the converse, of course, since an object could come to exist during a subinterval τ' of τ that does not exist throughout all of the larger interval τ.

The interpretation function V in a temporal structure needs to be revised slightly also. One of the most salient features of temporal processes is that things *change* over time. This has generated a venerable philosophical problem: how can an object be different at one time than at another and still be the very same object? Greek, medieval, and some contemporary philosophers put things in terms of substance and accident: the same substance can nonetheless alter those *accidental* properties that are not essential to its being that very substance. For example, Quayle's height is not essential to him, and hence it can change over time without Quayle ceasing to be Quayle. The same cannot be said of Quayle's being a human being. That property is essential to him; he couldn't come to be a stone, an alligator, a daisy, or otherwise come to lack it and still be Quayle.[12]

However we want to view the metaphysics of change, though, it is undeniable that our ordinary conceptual scheme—as reflected in the sort of ordinary language reports whose content IDEF3 is intended to capture—permits one and the same object to have different properties over time. We implement this in our formal semantics by relativizing the interpretation of predicates to intervals. Specifically, for a given n-place predicate P, and any interval $\tau \in T$, we let $V(\mathsf{P}, \tau)$ be a subset of D. In this way, an object $o \in D$ might be in the interpretation of P during one interval τ, i.e., we might have $o \in dom(\tau)$ and $o \in V(\mathsf{P}, \tau)$, and not be in the interpretation of P during another interval τ', i.e., we might also have $o \in dom(\tau')$ and $o \notin V(\mathsf{P}, \tau')$.

We do not require, analogous to the condition on *dom* above, that if an object has a property over a given interval, then it has that property over every subinterval. For while this is true of many properties, e.g., running, it is not true of all. Obvious cases are ones that involve some sort of average

[12]The concepts of substance, essence, and accident have in recent years generated a voluminous literature, much of it spawned by Kripke's formal work in the semantics of modal logic. For a good introduction to the issues, see (Schwartz, 1977.)

measurement, e.g., a manufacturing system over a given twenty-four hour period might have the (average) property of putting out fifty-two jobs per hour, while it might be that in no single hour subinterval of that period were there actually exactly fifty-two jobs put out. It thus has to be added specifically for each predicate in an IDEF3 representation whether or not its interpretation at a given interval τ is to be nested within each subinterval of τ.

We also do not require that the interpretation of a predicate at an interval contain only objects that "exist" at that interval. This is because there are many meaningful predicates that seem to contain objects that no longer exist. For example, Lincoln is in the extension of the predicate 'FORMER PRESIDENT'. Or suppose that in the course of constructing a given widget W one needs first to use a certain gadget G that is destroyed in the process before W is complete. After its construction, however, at a certain time τ one might want to be able to list which parts played a role in the construction of W, and hence one might want some sort of predicate '*PART USED IN CONSTRUCTION OF W*' that is true *at* τ of G, even though G no longer exists at τ, i.e., G $\notin dom(\tau')$. Of course, for certain purposes one might for one reason or another wish to enforce the condition that the interpretation of a predicate at an interval only take objects that exist during that interval, and such a condition could of course be added unproblematically, but for greater generality we omit it.

Interpretations for constants are given just as before: for any constant c, $V(\mathsf{c}) \in \mathcal{D}$. Assignment functions also work as before, assigning arbitrary objects in D to free object variables, and arbitrary intervals in TI to free temporal variables.

2.3.4 Truth and Realization in Temporal Structures

Atomic Formulas and Connectives Truth in temporal structures is now just a simple extension of truth in ordinary first-order structures. Specifically, let I be a temporal structure $\langle D, TI, dom, V\rangle$, and α an assignment for I. Then we define truth_α just as before, only relative to our interval indices: in the simplest case, an atomic formula Pc is true_α in I relative to τ just in case the object that c denotes, i.e., $V(\mathsf{c})$, is in the set $V(\mathsf{P}, \tau)$ of things that have the property expressed by P at the interval τ; if instead we have a variable x instead of c, then we look instead at $\alpha(\mathsf{x})$. Similarly for atomic

formulas constructed from n-place predicates. Connectives work as before, modulo the relativization to intervals.

Quantified Formulas Quantifiers present two options. (Again, discussion of quantifiers can be omitted without impairing the reader's comprehension of anything that follows.) On the one hand, given a quantified sentence $\exists \mathsf{x}\varphi$, we can interpret the quantifier as ranging over all objects in the domain D independent of the interval τ at which we are evaluating the formula, or we can relativize them to those objects in the domain of τ. Since the former is more general,[13] we will restrict our attention to it. Thus, $\exists \mathsf{x}\psi$ is true_α at τ just in case there is some object $o \in D$ and assignment function α' that differs from α at most in that it assigns o to x, such that ψ is $\text{true}_{\alpha'}$. That is, in essence, $\exists \mathsf{x}\psi$ is true_α in M at τ just in case ψ is true_α in M for some value in the domain D of all objects that α' assigns to x (while keeping all other variable assignments fixed). Similarly, $\forall \mathsf{x}\psi$ is true_α in M at τ just in case ψ is true in M for all values in D that we assign to x (while keeping all other variable assignments fixed).

2.3.5 Truth and Realization

As before, truth for a formula φ at an interval τ in an interpretation I is just for φ to be true_α at τ for all assignments α. Otherwise put, for a formula to be true at an interval in an interpretation is for it to hold during that interval no matter what values of the domain are assigned to the free variables of the formula.

Given all this we can define a notion analogous to truth at an interval for elaborations. Specifically, we say that an assignment α *realizes* an elaboration $E = [\mathsf{i}, \{\psi_1, \ldots, \psi_m\}, \{\varphi_1, \ldots, \varphi_n\}]$ in I just in case each ψ_i and each φ_j is true_α at $\alpha(\mathsf{i})$.[14]

[13] In particular, one can achieve the effect of quantification just over the objects that exist at the index at which one is evaluating the formula in question simply by introducing a distinguished predicate $E!$ whose extension at τ is always just the set of objects that exist at τ, i.e., $V(E!, \tau) = dom(\tau)$, for any τ.

[14] In a somewhat fuller development we would add a second type of elaboration that requires only that the formulas φ_i be true at some *sub*interval of τ. This enables one to capture descriptions that are less than precise about what *exactly* goes on during some period of time.

As it happens, in IDEF3, the notion of an assignment realizing an elaboration will play a much more prominent role than the notion of an elaboration simply being realized, and hence the notion of truth under an assignment will play a more prominent role than straight truth. The reason for this is that, first, different variable assignments for the same elaboration provide a natural representation of the idea of different objects instantiating the same process across time, and second, assignment of the same object to the same variable across different elaborations provide a natural representation of the flow of a given object through a process. These are the chief ideas for which IDEF3 is intended to be a flexible and powerful representation tool.

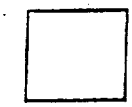
3 IDEF3 Graphical Syntax

3.1 Prediagrams and Their Grammar

We now turn to the more explicit development of IDEF3 proper. In addition to the (supplemented) first-order component already noted, the syntax of IDEF3 also has a graphical component, used for constructing figures that are vivid and especially useful for real world applications. There are several types of basic elements of the graphical syntax for IDEF3: *boxes, labels, arrows, and junction symbols.* With the exception of junction symbols, of which there are two, & and X, there is an inexhaustible (i.e., countably infinite) supply of elements of each type. Boxes and labels join to form *labeled boxes*, or *l-boxes*, and boxes and junction symbols join to form *junctions*—&-junctions and X-junctions respectively.[15] The preferred two dimensional (2-d) representations of these constructs are depicted in Figure 1.

Both labeled boxes and junctions are called *nodes.* Nodes are joined with other nodes to form what we call *prediagrams.* The joining of one node a to another a' by an arrow r can be represented as a triple $\langle a, r, a' \rangle$. r is called an *outgoing* arrow of a, and an *incoming arrow* of a'. The natural way to represent $\langle a, r, a' \rangle$ two dimensionally is to simply to draw the 2-d representation of r from the 2-d representation of a to the 2-d representation a'. Henceforth, we will often not distinguish between graphical elements of

[15] In practice, labels serve as concise abbreviations or descriptions of the state of affairs described by an elaboration associated with the box in a graphical diagram. For formal purposes we simply use lower case letters with subscripts; much more on this below.



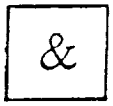
Box

$a_1, a_2, \ldots, a_n$

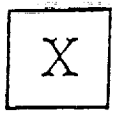
Labels

Labeled box
(A stands for any label)

Arrow



&-junction



X-junction

Figure 1: 2-d Graphical Lexicon

IDEF3 syntax and their 2-d representations, and we will often speak informally, e.g., of "drawing" an arrow from one node to another, of two nodes being "connected" by an arrow, of "connecting" an arrow to a node, etc. In particular, our rules below for constructing diagrams—mathematically, these are *graphs* of a certain sort—will be stated in these more informal (but no less rigorous) terms.

Not all ways of drawing arrows between nodes are legitimate prediagrams. Indeed, most ways of doing so are not. Most yield diagrams which are semantically unwieldy at best, and incoherent at worst. We impose order on the construction box and arrow figures out of our graphical syntactic elements by means of the following *recursive* definition of notion of a prediagram, i.e., a definition that begins with basic instances of the notion, and then proceeds to define more complex instances in terms of less complex. We will state the syntactic rules first; detailed explanation of each rule follows.

1. For $n > 2$, the result of drawing n arrows from an &-junction (resp., X-junction) to n distinct l-boxes is called an *basic open &-split* (resp., *basic open* X-*split*. (See Figure 2.)

2. The result of drawing an arrow from all the l-boxes in a basic open &-split (resp., basic open X-split) to a single &-junction is a *closed &-split* (resp., *closed* X-*split*). (See Figure 2.)

3. l-boxes, open and closed splits (basic or not) are *prediagrams.*

4. The l-boxes and junctions of a prediagram are called its *nodes.* A node in a prediagram with no outgoing arrows is called *extensible.*

5. A *path* in a prediagram is a sequence $\langle a_1, \ldots, a_n \rangle$ of nodes such that for $i < n$, there is an arrow from a_i to a_{i+1}. We say in this case that $\langle a_1, \ldots, a_n \rangle$ is a path *from* a_1 *to* a_n. Sequences with a single element are to be considered limiting cases of paths. A node a' in a prediagram π is *accessible from* a node a of π iff (i.e., if and only if) there is a path from a to a'. A path p is said to *traverse* a node a iff a is an element of p. a and a' are *basically incomparable* iff either (i) there is no path from a to a' or from a' to a, or (ii) there is a junction j such that both a and a' are accessible from j but every path from one to the other

traverses j.[16] a is *essentially accessible from* a' iff a is accessible from a' and a and a' are not basically incomparable.

6. A node p in a prediagram π is a *leftmost point*, or *L-point*, of π iff every node of π is accessible from p. p is a *rightmost point*, or *R-point*, of π iff p is accessible from every node of π. π is *L-pointed* if it has a unique L-point, *R-pointed* if it has a unique R-point, and *closed* if it is both L-pointed and R-pointed.

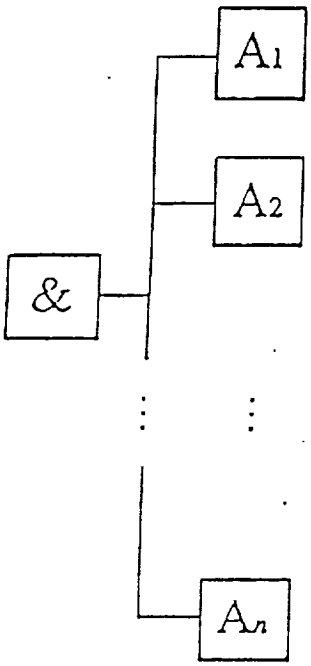
7. The result of replacing any l-box b in a closed split (of either sort) with a closed prediagram π by attaching the arrows coming into b to the L-point of π and the arrow coming out of b to the R-point of π is a closed split. It follows from this rule that every closed split is closed in the above sense. The area between the L-point and R-point of a closed split S is called the *scope* of S. If a is the L-point of S and b the R-point, then b is said to be a's *R-counterpart*, and a b's *L-counterpart*.

8. The result of drawing an arrow from an extensible node of a prediagram F to the L-point of another prediagram is itself a prediagram.

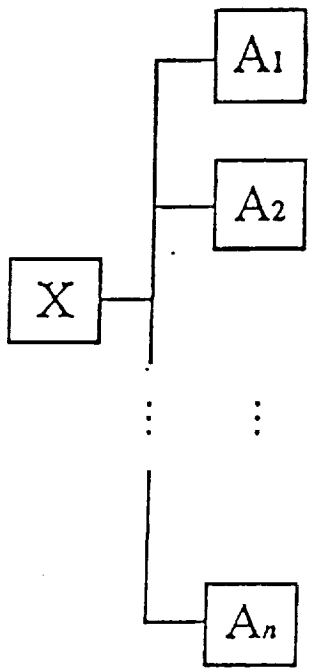
9. The result of drawing an arrow from an extensible node a of a prediagram π to any node a' of π is itself a prediagram iff (i) a is essentially accessible from a', (ii) no part of the arrow is within the scope of a closed split, and (iii) a' is neither the L-point of π nor the R-point of a closed split within π.

Further discussion of these rules will help make their function clear. As noted, the above definition is recursive in that it begins by introducing the basic cases of certain notions (prediagram, closed split) and then uses further rules to extend the notions once we have the basic cases. (1) and (2) give us some further initial elements to help get things started. Open &-splits represent processes (both types and instances) that diverge into several distinct subprocesses, and open X-splits represent process types that have a "conditional branch," i.e., a point where the process can flow one and only

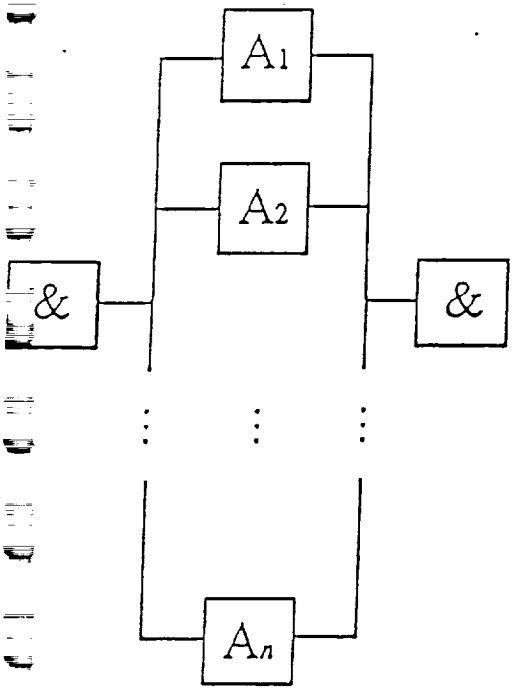
[16] The second condition here actually includes the first as a vacuous case, but this way of putting it makes the idea a bit clearer. Note that paths between basically incomparable nodes will be made possible only by (9) below, which provides for the construction of cycles within diagrams.



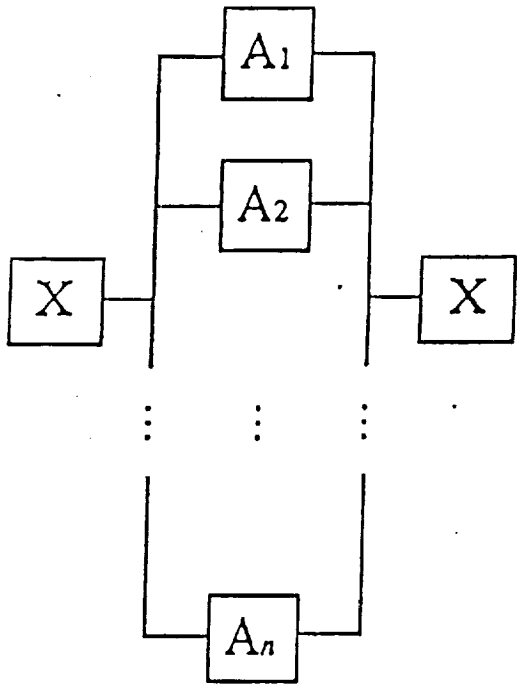
Open &-split



Open X-split



Closed &-split



Closed X-split

Figure 2: Open and Closed Splits

one of several ways. Such splits are obviously central to the description of indeterminate cycles within process types. A process instance generally flows one way rather than another at a branching point depending on whether or not some condition is met. If it is not, the process instance loops back to an earlier point in the process type, eventually returning to the branching point. When the satisfaction of the relevant condition cannot be determined in advance, the cycle in the process is indeterminate.

The difference between open &-splits and closed &-splits is that the branches of a process represented by the former are not conceived ever to converge at some later point back into a single stream while in closed &-splits they are. In and of themselves there is no essential difference between open X-splits and closed ones, since in both cases there can only be a single process that emanates from the branching point. However, in the context of a larger diagram there is a crucial difference. For only closed X-splits can be used to place conditional branches in the midst of a larger process that is to be construed as a single stream. An open X-split represents two possible wholly divergent paths a process can take; a closed X-split represents a mere option to jog one way or another on the path to a single end.

(3) together with (1) and (2) actually give us our base case for the recursion. (Compare the basic splits with atomic formulas in the grammar for first-order languages.) Note, however, that the definition is not restricted only to *basic* splits. This allows us to count other, more complex sorts of splits to be defined in (7) as well. (4)-(6) define some important auxiliary notions. (4) defines the notion of an extensible node, and (5) some useful graph theoretic concepts, and the notion of essential accessibility needed in (9).

Regarding (6), L-points and R-points in a prediagram intuitively represent definite beginning and ending points of a described process or subprocess. If some event is in fact a beginning point of a process, then everything that follows in that process can be traced back to that beginning point; this idea is captured in the requirement that all points in the prediagram—representing the events that make up the process—be accessible from the L-point. Analogously, if an event marks a definite end point to a process, then one should be able to trace back from that point to all preceding events in the process; thus the requirement that an R-point be accessible from all other nodes in the prediagram.

(7), (8), and (9) are the constructive parts of the definition; they define

complex prediagrams in terms of less complex parts. Specifically, given the notion of a closed prediagram, (7) gives us a recursive rule for generating more complex closed splits from properly closed prediagrams and less complex closed splits. The rough idea is that if a description represents a process that splits, then any of the descriptions of the subprocesses that branch off can be replaced by still more complex descriptions. (7) also defines the important notion of the *scope* of a split, to which we'll return shortly.

(8) and (9) tell how to build more complex prediagrams by drawing new arrows. (8) allows one to draw an arrow from an extensible node in one diagram to the L-point of another, a natural way in which one might put together prediagrams constructed from descriptions of different parts of a single system. The explicit purpose of (9) is to provide for the construction of diagrams that represent processes with (determinate or indeterminate) cycling. As noted above, this can only be done by drawing an arrow from some extensible node in a prediagram to some "preceding" box. This is captured by the requirement that the l-box a out of which the arrow is drawn must be accessible from the l-box a' to which the arrow is drawn. Consider for example a process that might be described as follows:

> *After its construction, a car body enters the painting area where it activates the paint-jets. It moves through the area and receives a coat of paint. Then it enters the drying area where it activates the dryers, and stays there for 20 minutes until it is dry. At that point, sensors check to see if the car body has received enough paint. If it has it is shunted off down the line; if not, it is put through the paint-dry cycle again, and continues in the cycle until it receives enough paint.*

To capture this description using the above syntax, one first draws a box to represent the construction of a car body. Since (let us suppose) we are not at present interested in the details of the construction, a single l-box labeled "Construct Car Body" (CCB for short) is used to represent that process. A second box labeled "Enter Paint Area" (EPA) is then introduced with an arrow (using (8)) drawn from the first box to the second. Three more boxes labeled "Paint Car Body" (PCB), "Enter Drying Area" (EDA), and "Dry Car Body" (DCB) respectively in the same fashion, each connected to the previous box by an arrow (by (8) once again). The two possible

outcomes of checking the paint are represented by an open split involving an XOR junction, with one box of the split labelled "Enough paint" (EP) and another labeled "Not Enough Paint" (not-EP). This open split can now be joined to the DCB box by (8) once again. A further box is introduced along the top branch of the split to represent (or summarize) the process that a car body enters upon completing the paint/dry cycle. All that remains is to represent the loop back into the cycle from the not-EP l-box. This is allowed by (9) since the not-EP box is essentially accessible from the EPA box, and the desired arrow is not within the scope of a closed split, nor connected to "illegal" l-boxes.

The condition that arrows cannot be drawn into or out of the scope of a closed split stems from the idea that subprocesses emanating from the L-point of a closed split should be *isolable*, i.e., that the only way in or out of a process represented by a closed split is from its beginning or end, respectively. If it should happen that a subdiagram within a closed split also accurately pictures a certain distinct subprocess outside the process represented by the split, then in constructing a prediagram one should simply copy the relevant section where it is needed in the prediagram, rather than to cross a split's logical boundaries.

The prediagram resulting from the above construction is shown in Figure 3. (The box labeled "Decomposition of CCB" dangling off the CCB box serves as a pointer to its more detailed meaning, or *decomposition*; this will be discussed below.)

The requirement (9) that a be *essentially* accessible from a' is added to rule out the use of (9) to generate certain pathological cycles that would arise if we were to allow arrows between l-boxes a, a' such that a is accessible from a' only because of a loop—generated by a previous application of (9)—from a, say, back to another l-box a'' from which both a and a' are accessible.[17] That a' cannot be the R-point of a closed split, as required by (iii), stems from the fact that such nodes mark the end of a split process, and hence should only have arrows coming into it from within the split. That it cannot be the L-point of the diagram represents the idea that processes typically have a unique starting state—represented by the L-point of the prediagrams that represent them—and hence it should not be possible to loop back to

[17] Such pathological cycles deserve much more discussion, but it would take us too far from the main point of this paper to do so here.

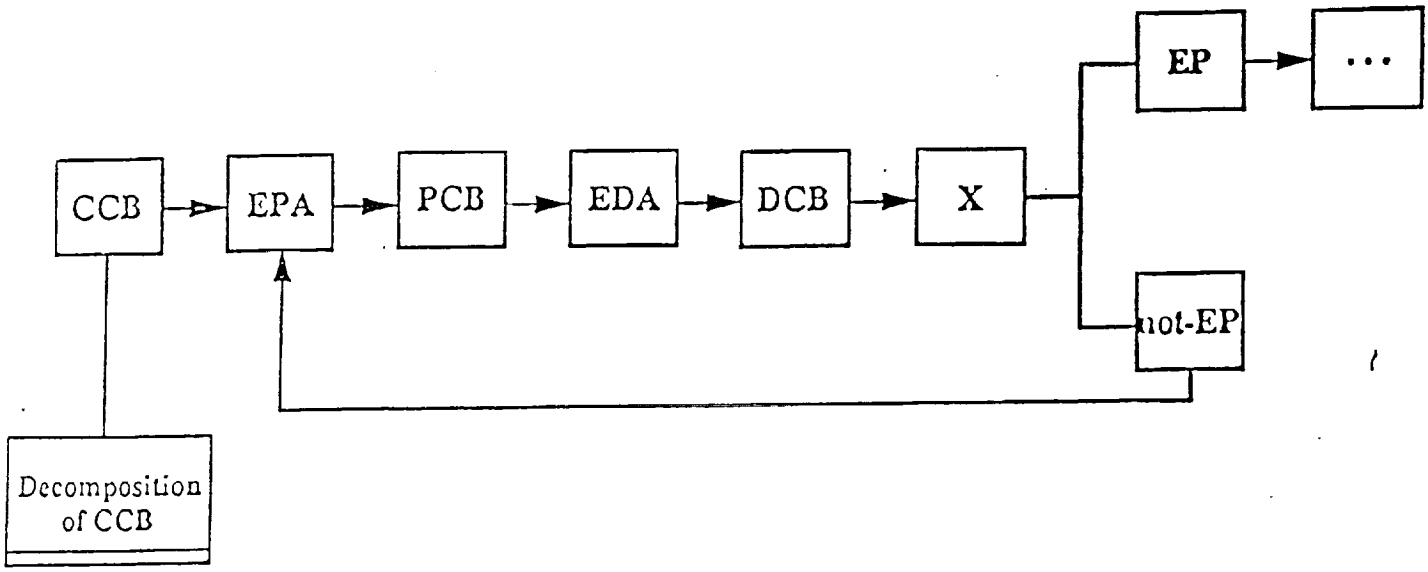


Figure 3: Prediagram for Paint/Dry Cycle

such states once the process has begun.[18]

Several further remarks are in order. First, except for the "target" node in the construction of a cycle, no node in a prediagram has more than one incoming arrow. Initially, this might lead one to believe that IDEF3 is incapable of representing convergent processes, where several processes in parallel converge to a single event. In IDEF3, however, convergence is to be represented by the convergence of several arrows to a single &-junction at the end of a closed split. In representing convergence by a closed split, the presupposition here is that all parallel convergent processes have a common initial path, that tracing back down each parallel process will reveal a common origin from which they all split—perhaps only the signing of a purchase order, or the system timer striking 8:00 a.m., but a common origin nonetheless. For otherwise there seems no initial ground that "parallelizes" the processes that converge in the first place.

Second, a number of constructs that will appear in IDEF3's more practical guise have been omitted here. For example, a common junction is *inclusive* OR, whereas we have included only *exclusive* OR. The reason for

[18] It should be noted though that this condition is dispensable.

this omission is that inclusive disjunction can be *defined* in terms of the other two.[19] Thus, *for theoretical purposes* it is redundant. In practice, of course, inclusive OR is very convenient, indeed, perhaps essential, since, e.g., it cuts down greatly on the size of certain prediagrams. It is also the most natural meaning of many ordinary uses of the word 'or' in common process descriptions, and hence is needed to capture such descriptions in the most natural way.

A second construct that for convenience we are also omitting are "relational links." These are links which can encode a wide variety of temporal and other information about events within a given process. These, like inclusive disjunction, are easily added to the theory.

Third, along the same lines, we note that further refinements on junctions can be defined that are useful in practice. For instance, it is possible to distinguish between *synchronous* &-junctions, where all the events represented by a given split must occur simultaneously, and *asynchronous* &-junctions, where this is not required. Since these are all definable in terms of our current apparatus, it will be omitted to avoid needless complexity. (These junctions and links, and more besides, can be found in the "user-oriented" version of IDEF3 sketched in the IDEF3 technical report.)

Finally, we note that we have made certain decisions that take form in corresponding restrictions on the syntax. It should be noted that there is a great deal of flexibility here, and that, within limits, alternative choices and views could be embodied in alternative definitions of the syntax.

3.2 Elaboration Tables and Decompositions

In this section we bring together our first-order syntax with our graphical syntax. As noted above, the label on a box is just an abbreviation for or terse description of (the content of) an associated elaboration. Theoretically, labels are superfluous; we could just as easily attach elaborations to boxes directly. But in practice, where one actually *uses* the 2-d graphical syntax, since there are no limits placed on the complexity of elaborations, there needs to be a more concise way of identifying the state of affairs a box represents. This is the role of the labels. The boxes in a prediagram are

[19] Specifically, an OR-split can be defined in terms of an X-split, with each branch leading to a possible conjunction of all the branches of the OR-split that could occur.

associated directly with elaborations via what we call an *elaboration table*—essentially, a one-to-one mapping from boxes to elaborations. In practice, an elaboration table might take any one of a number of forms depending on the relevant implementation of IDEF3: a pencil and paper implementation might use an actual written table on a separate sheet, whereas in a graphic-based computational implementation, boxes might take the form of a data structure one of whose elements is a pointer to an associated elaboration. However it is done, the point is that both labels and elaborations will appear in any implementation of IDEF3. Hence, since the theoretical cost of retaining labels in the theory is negligible, we thought it best to keep the formal theory more in step with practice at this point.

Formally then, where π is a prediagram, an *elaboration table* τ *for* π is a function that maps each l-box of π to an associated elaboration.

Let us illustrate these ideas by taking the paint-dry cycle prediagram above to the next stage in the construction of a complete IDEF3 diagram. Since we are for the moment suppressing the details of the actual construction of a car body when our focus is on the paint/dry cycle *per se*, the elaboration of the CCB box might be no more than $[i_1, \{\}, \{\text{Constructed}(x)\}]$. (The empty braces indicate that (as yet) no conditions have been placed on the temporal intervals during which the car body construction process occurs.) The EPA box's elaboration would be something like $[i_2, \{\}, \text{On}(x, \text{CB-carrier}), \text{Enters}(x, \text{paint-area}), \text{Activates}(x, \text{paint-jets})\}]$, and that of the PCB box something like $[i_3, \{\}, \{\text{On}(x, \text{CB-carrier}), \text{Moving}(x), \text{Activated(paint-jets)}, \text{Being-painted}(x)\}]$. The EDA box's elaboration, analogous to the EPA's, might be $[i_4, \{\}, \text{On}(x, \text{CB-carrier}), \text{Enters}(x, \text{drying-area}), \text{Activates}(x, \text{dryers})\}]$, and that of the DCB box might be $[i_5, \{\text{20-minutes}(i_5)\}, \{\text{On}(x, \text{CB-carrier}), \neg\text{Moving}(x), \text{Activated(dryers)}, \text{Drying}(x)\}]$; that of the EP box might be $[i_6, \{\}, \{\text{On}(x, \text{CB-carrier}), \neg\text{Moving}(x), \text{Activated(paint-sensors)}, \text{Value-of(paint-sensors,1)}\}]$, and that of the not-EP box $[i_6, \{\}, \{\text{On}(x, \text{CB-carrier}), \neg\text{Moving}(x), \text{Activated(paint-sensors)}, \text{Value-of(paint-sensors, 0)}\}]$; and as with the process prior to the paint/dry cycle, the process that continues after the paint/dry cycle is suppressed and summarized by a single box. This possible elaboration is depicted in tabular form in Figure 4.

The notion of suppression of detail in a diagram can be cashed in a rigorous way in IDEF3. The box noted above attached to the CCB box serves as a pointer to another prediagram/elaboration table combination $\langle \pi', \tau' \rangle$, one which, turns up the power of our descriptive microscope on the CCB

CCB: [t_1, {}, {Constructed (x)}]

EPA: [t_2, {}, {On(x, CB-carrier), Enters(x, paint-area), Activates(x, paint-jets)}]

PCB: [t_3, {}, {On(x, CB-carrier), Moving(x), Activated(paint-jets), BeingPainted(x,)}]

EDA: [t_4, {}, {On (x,CB-carrier), Enters(x, drying-area), Activates(x, dryers)}]

DCB: [t_5, {20-minutes}, {On(x, CB-carrier), ~Moving(x), Activated(dryers), Drying(x)}]

EP: [t_6, {}, {On(x, CB-carrier), ~Moving(x), Activated(paint-sensors), Value-of(paint-sensors, 1)}]

not-EP: [t_6, {}, {On(x, CB-carrier), ~Moving(x), Activated(paint-sensors), Value-of(paint-sensors, 0)}]

Figure 4: Elaboration Table for Paint/Dry Cycle Prediagram

box and represents its contents in greater detail. This is (roughly) what we refer to as a *decomposition* of the CCB box. A possible decomposition prediagram for the CCB box is pictured in Figure 5.

Decompositions, along with prediagrams and elaboration tables, are the final ingredient of full-blown IDEF3 diagrams. Informally, an IDEF3 diagram will consist of three elements: a prediagram π, an elaboration table ε, and an *decomposition function* δ; the diagram itself can thus be thought of as a triple $\langle \pi, \varepsilon, \delta \rangle$ consisting of such elements. A decomposition function for an IDEF3 diagram is a *partial* function—i.e., a function that is not necessarily defined everywhere in its domain—on the l-boxes of π that takes each box on which it is defined to another IDEF3 diagram.

We do not want the decomposition of any given box in a prediagram δ to contain that very box, or for that matter any other box in δ, or any of the boxes in *its* decomposition (if it has one). A decomposition, after all, is a closer look at some event within a larger process; hence it seems quite reasonable that no smaller part of that event could contain within itself the original event, or any other part of the larger system. This conception of processes in fact comports well with the way one would draw decompositions if one were using a large sheet of paper: each decomposition would consist of entirely new boxes drawn below the decomposed box of the original diagram.

At the same time, we do want to allow "nested" decompositions. That is, we always want it to be possible to raise the power of our descriptive microscope even farther than we have at a given point in the course of description capture. Thus, we want to allow boxes within decompositions themselves to admit of further decomposition, and boxes within those decompositions to admit of yet further decomposition, to any desired finite depth.

To achieve all this in a rigorous fashion, we proceed as follows. First, since the set B of boxes is infinite, we can divide it up into infinitely many mutually disjoint infinite sets B_0, B_1, etc. (Labels, as noted, are theoretically superfluous, and can be ignored in the present context.) We will use the first of these sets, B_1, to define an initial (base) set $diag_0$ of *basic* IDEF3 diagrams all of whose decomposition functions are everywhere undefined; no box in a diagram at this stage, that is, has a decomposition. (Recall that decomposition functions are partial, and hence need not be everywhere, or even anywhere, defined.) At the next stage we build a new set $diag_1$ of IDEF3 diagrams out of the boxes in B_2, only now we allow them to contain decomposition functions that take l-boxes to diagrams in the base set $diag_0$.

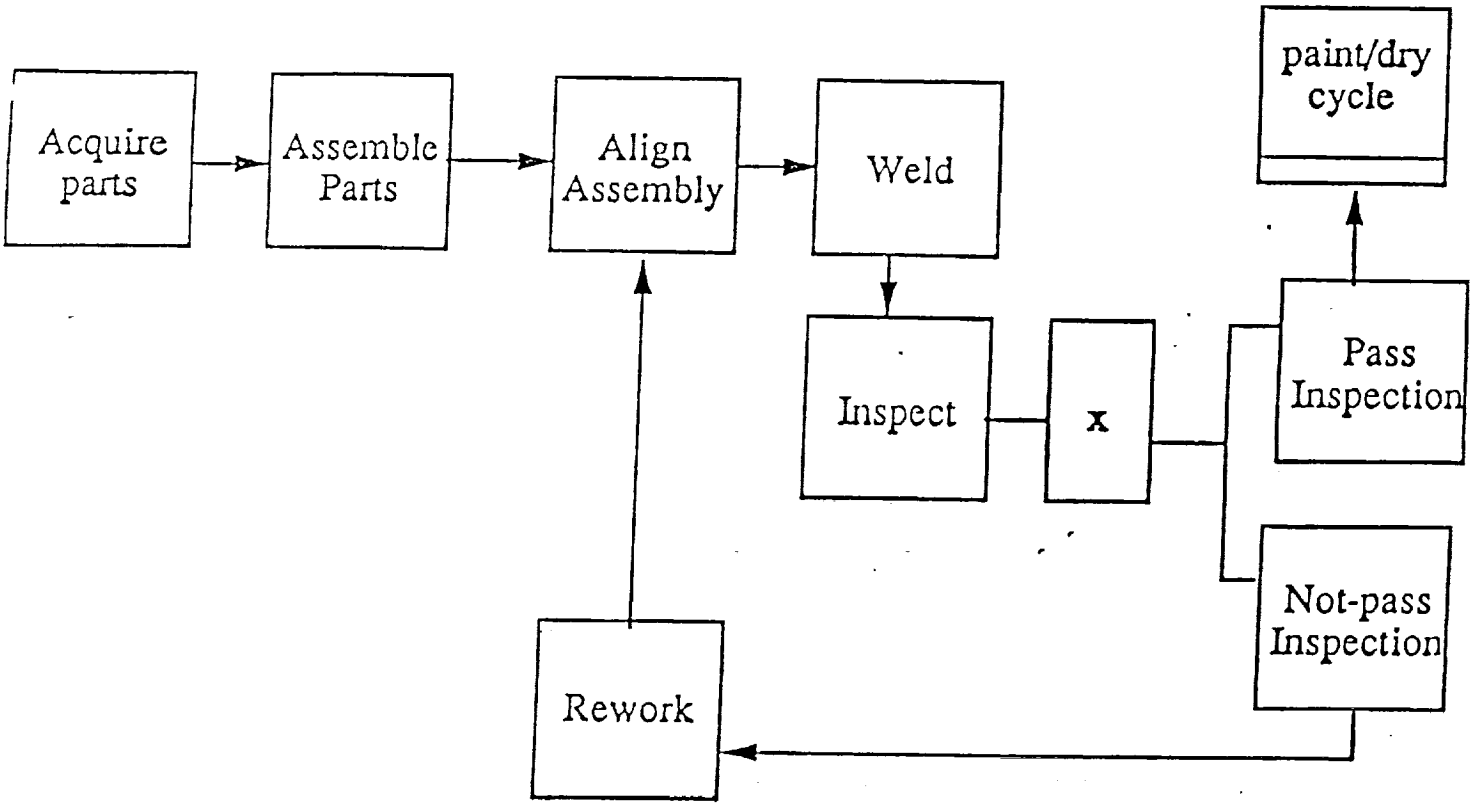


Figure 5: CCB Box Decomposition Prediagram

This will enable us to satisfy the condition that the decomposition of a given box b in $diag_1$ cannot contain b itself, or any other box b' at the same "level" as b in the diagram.

This is not quite enough, though. For, as we are defining them here, boxes in IDEF3 diagrams all represent temporally nonoverlapping events, i.e., events which intuitively could have no common parts. Hence, we also require that the decompositions of any two distinct boxes b, b' of π must not contain any common box. (In fact, IDEF3 can be generalized to allow temporal overlap, but even in this case it turns out to be most convenient still to enforce this condition and represent even the common parts of overlapping events by distinct boxes in each decomposition.[20])

We must impose one further condition given the distinction between closed and nonclosed prediagrams. Let π be a prediagram constructed out of boxes in B_1. We cannot allow any given box b of π to be mapped to just any diagram in $diag_0$. For if b is nonextensible, i.e., if b has an outgoing arrow, then it can only be mapped to a *closed* diagram,[21] since the events represented by such boxes intuitively do not branch at their end but continue in a single stream. Speaking more syntactically, since a decomposition represents a closer look at a given event, we should always be able to replace any box in a diagram with its decomposition. The only way to do so in the case of a nonextensible box is to attach its incoming arrows to the L-point of its decomposition, and its outgoing arrow to the R-point of its decomposition. This can only be done if we impose the above condition.

Diagrams at successive levels $diag_n$, $n > 1$ are defined in just the same way. In general, that is, for $n > 0$, an *n^{th}-level diagram*, i.e., an element of $diag_n$, is a triple $\langle \pi, \varepsilon, \delta \rangle$ such that π is a prediagram whose boxes are in B_n, ε is an elaboration table for π, and δ is a partial function from the boxes of π to the diagrams in $diag_{n-1}$ satisfying the above conditions. The set of all diagrams is then defined to be the union of the $diag_m$, for $m \geq 0$. Our definition thus allows for the nesting of diagrams within diagrams to any finite degree.

A few brief remarks are in order. First, note that the condition that the decomposition function for an *n^{th}-level diagram* only take values in $diag_{n-1}$,

[20] This condition enables us to streamline the semantics considerably, since, as we will see, the definition of realization for prediagrams in $diag_0$ serves for all prediagrams generally.

[21] Where a diagram $\langle \pi, \varepsilon, \delta \rangle$ is *closed* just in case π is closed.

and not $diag_m$ for all $m < n$ is not at all restrictive since each level contains a "copy" of all previous levels.[22] Thus, there is no diagramatic structure available for decomposition functions at one level that is unavailable at successive levels.

Second, the reader may have noted similarities between the structure of IDEF3 diagrams and dynamic record types in high level languages like Pascal. In addition to a variety of standard fields, such record types can contain one or more fields whose value for a given record consists of a pointer to a record of the same type. In the same way, a decomposition function for a given IDEF3 diagram (applied to a box in the diagram) points to yet other diagrams. Unlike true dynamic record types, though, our hierarchical construction rules out the possibility of diagrams that point back to themselves.

Let us make our definition entirely formal for good measure. For a given graph π constructed out of the elements of IDEF3 graphical syntax, let $boxes(\pi)$ be the set of boxes that occur in π, let $PD(\pi)$ mean that π is a prediagram, and let $ET(\varepsilon, \pi)$ mean that ε is an elaboration table for π. Where $a = \langle \pi, \varepsilon, \delta \rangle$, let $(a)_1 = \pi$. Then we have:

$diag_0 = \{\langle \pi, \varepsilon, \delta \rangle \mid PD(\pi)$ and $boxes(\pi) \subseteq B_0$ and $ET(\varepsilon, \pi)$ and $\delta = \emptyset\}$;

$diag_n = \{\langle \pi, \varepsilon, \delta \rangle \mid PD(\pi)$ and $boxes(\pi) \subseteq B_n$ and $ET(\varepsilon, \pi)$ and $domain(\delta) \subseteq boxes(\pi)$ and $range(\delta) \subseteq diag_{n-1}$ and $\forall x \in domain(\delta)$, if $not\text{-}EXTENSIBLE(x)$, then $CLOSED(\delta(x))$, and $\forall x, y \in domain(\delta)$, $boxes((\delta(x))_1) \cap boxes((\delta(y))_1) = \emptyset\}$, for $n > 0$;

$diag = \bigcup_{m \geq 0} diag_m$.

4 IDEF3 Semantics

We now want to give a semantics for IDEF3 graphs that will enable us to hook them up with all the more standard first-order apparatus that we developed above. An l-box in a diagram represents a (part of) process that is described by the associated elaboration $E = [\mathrm{i}, \{\psi_1, \ldots, \psi_m\}, \{\varphi_1, \ldots, \varphi_n\}]$.

[22] For example, since decomposition functions are partial and each B_i is infinite, $diag_1$ contains in particular all the diagrams that can be formed from the boxes of B_1 that, like the basic diagrams, have decompositions that are undefined everywhere. There will thus be an isomorphic copy in $diag_1$ of each basic diagram in $diag_0$, and similarly for all succeeding levels.

An *instance* of that process will exist if during the temporal interval represented by i (i.e., if at the value of i under some assignment) all the sentences ψ_i, φ_j are true. As noted already, the use of variables is crucial to this idea, since object variables can have different objects assigned to them as values, and temporal interval variables different temporal intervals. This enables one to instantiate the same elaboration in many different ways, thus capturing the different ways in which a general process might be realized. We make these ideas more formal in this section.

4.1 Instantiation Graphs

Interpretations provide mathematical models of the general structure of a described process. Interpretations together with variable assignments enable one to model, or *realize*, specific instances of the general process.

Let us fix a given IDEF3 diagram $\langle \pi, \varepsilon, \delta \rangle$ written in some language $\mathcal{L}$. For any l-box $b \in boxes(\pi)$, let $\varepsilon(b)$ be the elaboration associated with b by the elaboration table ε. Given an interpretation I for $\mathcal{L}$, an $\mathcal{L}$-I-assignment α can be said to *realize* the l-box b just in case it realizes $\varepsilon(b)$.

Now, with every prediagram π can be associated a set (generally infinite) of associated graphs which characterize the possible ways in which the system that the prediagram describes can be instantiated. Thus, we call these associated graphs *instantiation graphs for* π. If π has no cycles (i.e., no paths that begin and end at the same node), then its instantiation graph (there is only one in this case (up to isomorphism)) is especially easy to characterize, since it will look just like π itself except with its junctions removed. That is, roughly, form the instantiation graph for π by beginning with the L-point of π and then tracing down from the L-point, add as nodes all the l-box "children" of any &-junction one comes to, and only one of the l-box children of any X-junction one comes to until one cannot continue. The procedure for generating instantiation graphs from prediagrams π with cycles is similar, except one must in addition allow an instantiation graphs to "unfold" the cycles some finite number of times (if possible; some legitimate prediagrams with unconditional cycles have only infinite instantiation graphs).

(It might be wondered why junctions are removed from instantiation graphs. The reason is that junctions are needed primarily because of the two ways in which a process might split that are recognized in IDEF3—conjunctive splits and exclusive disjunction splits. If there were only one

sort, then we could just as well have multiple arrows coming straight out of an l-box. In an actual process *instance* there is in fact only the first sort of splitting; where a diagram has an X-split, representing possible continuations of the process represented by the diagram, any corresponding instance of the process goes either one way or the other; there is no actual disjunctive splitting at the instance level. The splitting is only at the type level. Thus, in an instantiation graph, since there is only one kind of splitting—where the process flow diverges into several streams—junctions are superfluous.[23])

Despite the simplicity of these ideas, it takes a little work to give it formal expression. To begin, let $\Delta = \langle \pi, \varepsilon, \delta \rangle$ be an IDEF3 diagram, and let $\Gamma = \langle V, A \rangle$ be a graph such that all the vertices $x \in V$ are of the form $\langle a, n \rangle$, where a is a node of π, and n is a natural number greater than 0. As usual, A—the set of edges, or arcs, of the graph—is a set of pairs of members of V (hence a set of pairs of the form $\langle \langle a, n \rangle, \langle b, m \rangle \rangle$; intuitively, each pair represents an arc from the first member to the second). If a and b are nodes of π, then we say that b is a *successor* of a (in π), and a a *predecessor* of b, if there is an arc from a to b in π.

Now, say that Γ is π-*generated* iff, first, $\langle a, 1 \rangle \in V$ iff a is the L-point of π, and for all nodes a of π and all natural numbers n, if $\langle a, n \rangle \in V$ and a has

[23] Actually, for all we have said there is a further reason for retaining &-junctions. Our syntactic rules allow *nested* &-junctions (and X-junctions), i.e., junctions that are successors of other junctions. In these cases &-junctions can serve as grouping delimiters that might well put constraints on the relations between events that instantiate a diagram. For example, we might want to distinguish two cases where three events branch off from a given event: on the one hand, we might want to allow the three to begin at any three times after the given event; on the other hand, we might want to group two of the events and constrain matters such that the two must begin either before or after the third, and hence that they cannot "sandwich" the third temporally. The most natural way to group the two with our syntax would be to represent them as a small closed split whose L-point is, along with a box representing the third event, an immediate successor of another &-junction. But then, if junctions are eliminated in instantiation graphs, it seems that this grouping information would be lost. However, as already noted, the syntax of IDEF3 allows one to put additional constraints on temporal intervals other than the intervals represented by the dominant temporal interval of an elaboration. The chief reason for allowing this is to enable one to put constraints on the temporal intervals associated with the distinct branches of branching processes. Thus, one can simply transfer the grouping information carried by the nested &-junctions explicitly into the elaborations in terms of the appropriate conditions. (Note how this capability is captured in the semantics in Clause 4 of the definition of realization for instantiation graphs in Section 4.2.)

any successors, then

1. if a has a unique successor b, then

   (a) if b is a box, or the L-point of a split, or the R-point of a closed X-split, then $b \in A_{n+1}$ and $\langle\langle a,n\rangle,\langle b,n+1\rangle\rangle \in A$;

   (b) if b is the R-point of a closed &-split, then (i) $\langle b,n+j\rangle \in V$, where j is the least number such that for each of b's predecessors c, $\langle c,i\rangle \in V$, where $m < i < n+j$, where m is the largest number $< n$ such that $\langle b',m\rangle \in V$, where b' is b's L-counterpart, and (ii) $\langle\langle a,n\rangle,\langle b,n+j\rangle\rangle \in A$;

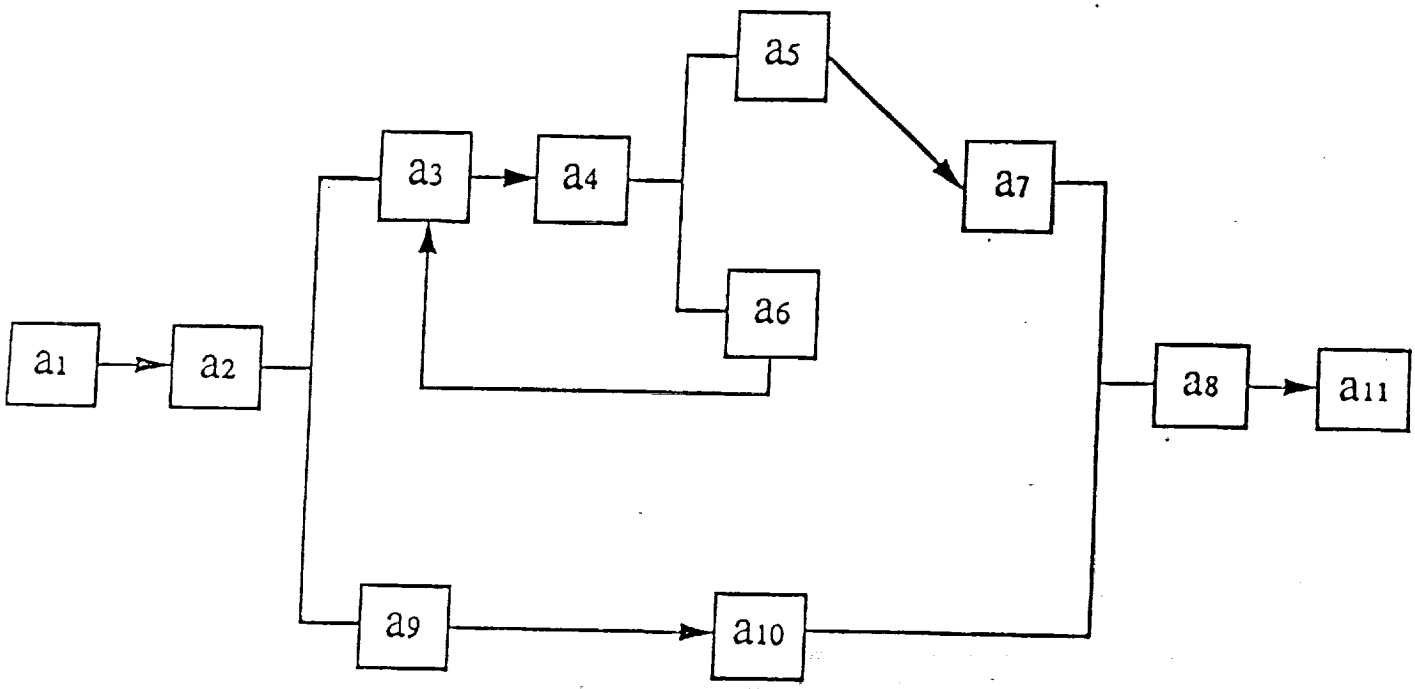
2. if a does not have a unique successor, then

   (a) if a is the L-point of an &-split, then all of a's successors are in A_{n+1}, and for each such successor b, $\langle\langle a,n\rangle,\langle b,n+1\rangle\rangle \in A$;

   (b) if a is the L-point of an X-split, then exactly one of a's successors b is in A_{n+1}, and $\langle\langle a,n\rangle,\langle b,n+1\rangle\rangle \in A$.

If a is not a junction, a vertex $\langle a,n\rangle$ in a π-generated graph represents an occurrence of an instance of the event represented by a in the prediagram π at a certain stage in a run of the system represented by the entire prediagram. The need for indexing a with a number n arises from the possibility of cycles in the system—different indices paired with the same box a represent different instances of the same event. The rather complex 1(b) in particular ensures that the R-point of a closed split is assigned the right index relative to its predecessors for a given possible run of the system.

We illustrate these ideas in Figure 6, where we have a prediagram π and an associated π-generated graph. For formal purposes we are ignoring labels and simply tagging the boxes of the prediagram directly (in a full fledged prediagram, of course, a_2, a_4 and a_8 would all have to be junctions). Note that π contains a cyclic X-split down one path of an &-split. This allows for indeterminate cycling from instances of box a_3 to instances of box a_6 and back again. The π-generated graph represents a process instance in which there is just one such cycle before the process flows from (an instance of) a_3 to a_7, and finally to a_{11}. Note that the two traversals of a_3 are distinguished by two different indices, 3 and 6 respectively.



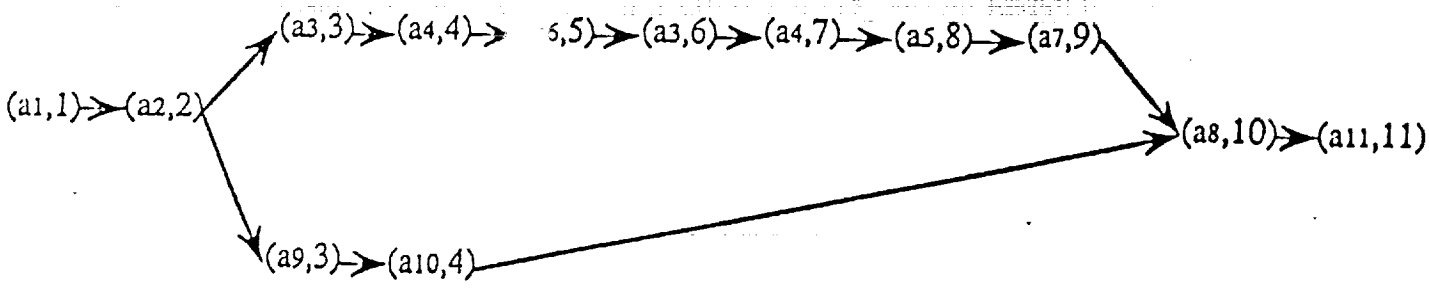


Figure 6: Prediagram π with π-generated Graph

The prediagram requires that instances down each path complete before there can be an instance of a_{11}. This means that we need to have arcs running from the "last" event down each branch to the same instance of a_{11}, in this case represented by $\langle a_{11},11\rangle$. So $\langle a_{10},4\rangle$ has to "wait" until the upper branch is completed before being connected to the close of the split a_8; we cannot, that is to say, index a_8 with the number 5 and define an arc from $\langle a_{10},4\rangle$ to $\langle a_8,5\rangle$, since $\langle a_8,5\rangle$ would indicate different point in time than does $\langle a_8,10\rangle$. This is assured by 1(b). To see this, let $\langle a,n\rangle$ in the definition be $\langle a_{10},4\rangle$, and let b be a_8 (the R-point of the above closed &-split; then the definition requires that for each of a_8's predecessors c—in this case only a_7 is relevant—if $\langle c,i\rangle$ is a node—in this case $\langle a_7,9\rangle$, then where j is the least number such that when added to n the sum is greater than any such i—so j is 6 in our example—$\langle a_8,4+6\rangle$, i.e., $\langle a_8,10\rangle$ must be a vertex of our π-generated graph, and there must be an arc from $\langle a_{10},4\rangle$ to $\langle a_8,10\rangle$. The number m in the definition—which is the number 2 here and plays no significant role—ensures in general that we are dealing with instances within the same subprocess represented by the entire closed split, since they too can be traversed more than once in a given large process. Such multiple traversals would be indicated by different indexed occurrences of a_2. The role of m is to ensure that we always are dealing with instances of events within an instance of the split that have occurred since the most recent traversal of a_2.

A π-generated graph $\Gamma = \langle V,A\rangle$ is said to be π-*admissible* iff it has no proper π-generated subgraph.[24] The reason for this definition stems from the fact that nothing in the idea of a π-generated graph rules out the possibility of all sorts of extra "junk" getting thrown into the nodes or arcs of such a graph over and above what is sanctioned by the definition alone. For instance, the sorts of graphs we are interested in are *simple*, i.e., they have at most one arc between any two given nodes; but, e.g., an extraneous arc can be added between any two nodes of a π-generated graph and the result will still be a π-generated graph. For the definition tells us only what must be present in such a graph, not what must *not* be present. Thus, such graphs at a given stage n do not necessarily represent the state of a given run of the system being represented by π at that point. We filter such unintended graphs out of consideration by focussing our gaze on only the "smallest" graphs that

[24] Where a graph $\Gamma' = \langle V',A'\rangle$ is a *subgraph* of a graph $\Gamma = \langle V,A\rangle$ iff $V' \subseteq V$ and $A' \subseteq A$. Γ' is a *proper* subgraph of Γ if it is a subgraph and either $N' \neq V$ or $A' \neq A$.

satisfy the definition, hence those with no extraneous nodes or arcs. The π-generated graph of Figure 6 is also π-admissible; it could be transformed simply into an inadmissible graph simply by, say, adding a new unconnected node $\rangle a_3, 17\langle$ to the set of vertices.

Given a π-admissible graph $\langle V, A\rangle$, our final task is to eliminate all the junctions from V and revise A in the obvious way to obtain the graph $\langle V^*, A^*\rangle$. That is, in the simplest case, where a and c are boxes of π and b is a junction, if $\langle\langle a,n\rangle, \langle b,m\rangle\rangle, \langle\langle b,m\rangle, \langle c, m+1\rangle\rangle \in A$ (that is, if there is an arc from $\langle a,n\rangle$ to $\langle b,m\rangle$ and from $\langle b,m\rangle$ to $\langle c, m+1\rangle$), then we remove b and the above arcs from V and A respectively, and define a new arc from $\langle a,n\rangle$ to $\langle c, m+1\rangle$, i.e., we add the arc $\langle\langle a,n\rangle, \langle c, m+1\rangle\rangle$ to A^*.

More formally, then, let $\Gamma = \langle V, A\rangle$ be a π-admissible graph. Let V^* be $V - \{\langle a,n\rangle \in V \mid a \text{ is a junction of } \pi\}$, and let A' be $A - \{\tau \in A \mid \tau \text{ contains a junction of } \pi\}$. Now let S be the set of all paths $\langle\langle a_1, n_1\rangle, \ldots, \langle a_m, n_m\rangle\rangle$ of Γ, $m > 2$, such that and a_1 and a_m are boxes of π, and for all i such that $1 < i < m$, a_i is a junction of π. So S contains all the paths of Γ between two boxes whose intervening nodes are all junctions. Let S' be the set of all pairs $\langle a_1, a_m\rangle$ such that $\langle a_1, \ldots, a_m\rangle \in S$, i.e., the set of all pairs consisting of the first and last elements of some path in S; S' is thus the result of deleting all the intervening junctions between the first and last boxes in a path in S. Let A^* be $A' \cup S'$. Then where Γ^* is $\langle V^*, A^*\rangle$, we say that Γ^* is an *instantiation graph* for π. The instantiation graph that results from removing the junction vertices from the π-generated graph of Figure 6 is illustrated in Figure 7.

4.2 Realizing Instantiation Graphs

Now we want to say what it is for an instantiation graph to be realized by a class of associated assignment functions (in a given interpretation). We will set up a correlation between each node of an instantiation graph and an associated assignment function for the elaboration associated with that node. For the assignment functions as a whole to realize the graph, they have to interpret it jointly in such a way as to represent a given run of the system; e.g., they have to map adjacent vertices to adjacent temporal intervals, they have to interpret identical variables in the corresponding elaboration tables of adjacent vertices by means of the same objects to capture object flow through the system, etc.

The apparatus is straightforward, but a bit involved. First we set up the

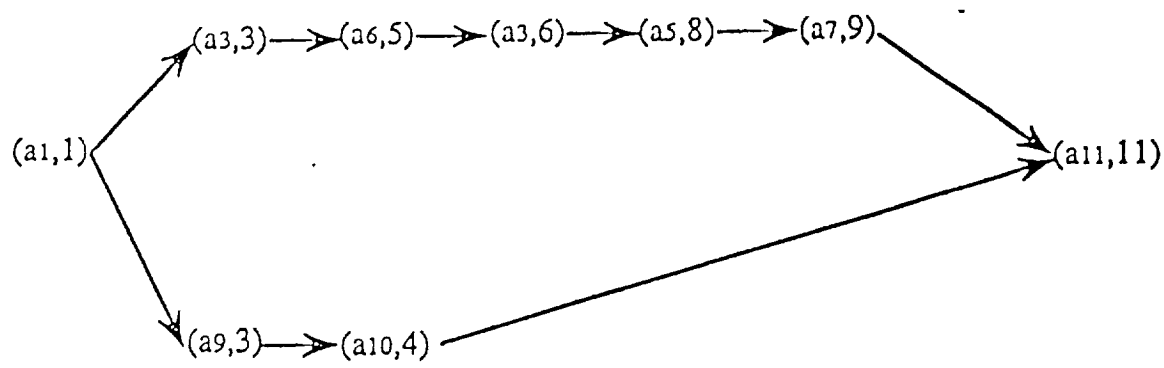


Figure 7: Instantiation Graph for π

relevant syntactic apparatus. Let $\mathcal{L}$ be a first-order language used of the sort defined above and $\Delta = \langle \pi, \varepsilon, \delta \rangle$ be an IDEF3 diagram whose language for its elaborations is $\mathcal{L}$, and let $\Gamma = \langle V, A \rangle$ be an instantiation graph for Δ. For a given box a, of π, let i_a be the dominant temporal variable of the elaboration $\varepsilon(a)$ associated with a. We shall assume for ease of exposition that $\varepsilon(a)$ contains no quantified formulas for every box a of π.

Thus, for the corresponding semantic apparatus, let I be an interpretation for $\mathcal{L}$, and let F be a mapping of the vertices in V to assignment functions such that, for all $\langle a, n \rangle \in V$, $F(\langle a, n \rangle)$ is defined on *only* the variables in $\varepsilon(a)$.[25] We write 'α^a_n' for the assignment $F(\langle a, n \rangle)$. For any such assignments α, α', we define '$\alpha \sim \alpha'$' to mean that α and α' agree on all the variables on which they are both defined.

Given all this, we say that F *realizes* Γ just in case, for all $\langle \langle a, n \rangle, \langle b, m \rangle \rangle \in A$,

[25] It is not only more convenient to use such "partial" assignments, but it also corresponds more closely to the way in which assignments would be implemented in a computational tool. In a more detailed treatment the notion of a partial assignment would be defined more explicitly.

1. α_n^a realizes $\varepsilon(a)$ and α_m^b realizes $\varepsilon(b)$;

2. $\alpha_n^a \sim \alpha_m^b$;

3. if $\langle b, m\rangle$ is $\langle a, n\rangle$'s only successor and $\langle a, n\rangle$ is $\langle b, m\rangle$'s only predecessor, then $\alpha_n^a(i_a)$ meets $\alpha_m^b(i_b)$;[26]

4. if $\langle a, n\rangle$ has more than one successor, then (i) for all its successors $\langle c, l\rangle, \langle c', l'\rangle$, $\alpha_l^c \sim \alpha_{l'}^{c'}$, (ii) there is one successor $\langle c, l\rangle$ such that $\alpha_n^a(i_a)$ meets $\alpha_l^c(i_c)$ and (iii) for all remaining successors $\langle c', l'\rangle$, $\alpha_n^a(i_a)$ meets or precedes $\alpha_{l'}^{c'}(i_{c'})$;

5. if $\langle a, n\rangle$ has more than one predecessor, then (i) for all its predecessors $\langle c, l\rangle, \langle c', l'\rangle$, $\alpha_l^c \sim \alpha_{l'}^{c'}$, (ii) there is one predecessor $\langle c, l\rangle$ such that $\alpha^c l(i_c)$ meets $\alpha_n^a(i_a)$ and (iii) for all remaining predecessors $\langle c', l'\rangle$, $\alpha_{l'}^{c'}(i_{c'})$ meets or precedes $\alpha_n^a(i_a)$;

(1) is required because, for F to realize Γ, the assignment it associates with each vertex must realize the corresponding elaboration. The reason behind (2) takes us back to our remarks on object flow. As noted, the same free variable occurring in the elaborations of connected boxes represent the flow of a single object from one event to another. Thus, we require that for F to realize an instantiation graph, assignments for adjacent vertices must agree on the free variables common to the boxes in those vertices. This is just what (2) requires.

(3) captures the idea that boxes connected by an arrow represent temporally contiguous events within a larger process. This is a rather stringent requirement adopted primarily for ease of exposition. In fact, we can generalize our apparatus and allow virtually any temporal relation to obtain between any two given boxes. Temporal contiguity serves here only as a convenient default.

Similar remarks apply to (4). Splitting in an instantiation graph represents the branching of a process into several parallel subprocesses. We have chosen as a default semantics that the event represented by the vertex that splits meet at least one of the branches, and that meet or precede all the

[26] Where, recall, an interval τ meets another τ' iff the end point of τ is the beginning point of τ'.

others. This is captured by 4(ii) and 4(iii). 4(i) places the additional requirement that any two of the assignments associated by F with the successors of the branching vertex agree on free variables on which they are both defined. This permits one in particular to place additional temporal constraints on the intervals occupied by the branches of branching processes, e.g., that one of the branches precede another, that two begin together and precede a third, etc.

(5) is essentially the "dual" of (4) for the "back end" of a splitting process that eventually converges.

We say that I realizes a basic IDEF3 diagram[27] $\langle \pi, \varepsilon, \delta \rangle$ iff, for any assignment function α on the variables of the elaboration $\varepsilon(p)$ of the L-point p of π that realizes $\varepsilon(p)$, there is an instantiation graph Γ and a mapping F from the vertices of Γ onto assignment functions such that (i) $F(p,1) = \alpha$, and (ii) F realizes Γ. (Intuitively, this captures the idea that any time the initial process of a system occurs, the entire system will be instantiated in some way.) Though we will avoid the details here, it should be quite clear that any nonbasic IDEF3 diagram Δ can in principle be "expanded" to a basic diagram Δ'—called its *basic expansion*—that explicitly includes all of Δ's nested decompositions.[28] We can thus say that I *realizes* an IDEF3 diagram Δ generally just in case it realizes its basic expansion.

5 Conclusion

This paper has provided a formal definition of the process description capture method (IDEF3) using an extended first-order logic. We began by distinguishing between descriptions (recordings of the beliefs about the world around us) and models (systems of objects and relations that describe a real-world system). We described the term "process" in its general sense as a *unit of behavior* in IDEF3. Next, we reviewed the fundamentals of first-order logic, and extended it to include temporal and index semantics. Finally, we described the IDEF3 notion of an *elaboration*, and formally specified the graphical syntax for the method.

[27] I.e., recall, a diagram with no decompositions.

[28] I.e., all the decompositions of the boxes in Δ's prediagram, all the decompositions of the boxes in those decompositions, etc.

This formalization was necessary for five reasons: (1) to identify the informal intuitions that motivate the method, (2) to provide a technical basis for integration with other methods, (3) to provide an objective basis for comparison with other methods, (4) to provide technical basis for the design of other methods, and (5) to provide accurate specifications for the design of automated IDEF3 support tools. This formalization, combined with the corresponding technical report and supporting documents, should fully describe the IDEF3 process description capture method.

References

BARWISE, J., 1989, *The Situation in Logic* (Stanford: Center for the Study of Language and Information).

BARWISE, J., AND PERRY, J., 1983, *Situations and Attitudes* (Cambridge, Mass: MIT Press/Bradford Books).

BLINN, T., MAYER, R., BODEMILLER, C., and COOK, S., 1989, Automated IDEF3 and IDEF4 system design specification. Interim Technical Report, Contract No. 055, NASA Cooperative Agreement No. NCC9-16, Project No. IM.15, NASA RICIS Program, University of Houston, Clear Lake, Texas, U.S.A.

CHELLAS, B., 1980, *Modal Logic: An Introduction* (Cambridge: Cambridge University Press).

COLEMAN, S., 1988, IDS (Integrated Design Support system) integrated system engineering process. Interim Technical Report, Pacific Information Management, Inc., Santa Monica, California.

CORYNEN, G. C., 1975, A Mathematical Theory of Modeling and Simulation. Ph.D. Dissertation, Department of Engineering and System Science, University of Michigan.

CURRY, G., DEUERMAYER, B., and FELDMAN, R., 1989, *Discrete Simulation: Fundamentals and Microcomputer Support* (Oakland: Holden-Day, Inc.).

FROST, R., 1986, *Introduction to Knowledge Base Systems* (New York, McGraw-Hill Publishing Co.)

GALLAIRE, H., and MINKER, J., 1978, *Logic and Databases* (New York: Plenum Press).

HUGHES, G. E., and CRESSWELL, M. J., 1968, *An Introduction to Modal Logic* (London, Methuen and Co. Ltd.).

JACOBS, B. E., 1982, On Database Logic. *Journal of the Association for*

Computing Machinery, **29**, 310-332.

KRIPKE, S., 1963, Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, **16**, 83-94.

LIN, M., 1990, Automated simulation model design from a situation theory based manufacturing system description. Ph.D. dissertation, Department of Industrial Engineering, Texas A&M University, U.S.A.

MAYER, R., 1988, Cognitive skills in modeling and simulation. Ph.D. dissertation, Department of Industrial Engineering, Texas A&M University, U.S.A.

MAYER, R., MENZEL, C., and MAYER, P., 1989, IDEF3 technical reference report. IISEE Program Interim Technical Report, AFHRL/LRL, Wright-Patterson Air Force Base, Ohio.

MAYER, R., BLINN, T., COOK, S., and BODEMILLER, C., 1990, Automated support tool for IDEF3 and IDEF4. Final Technical Report, Contract No. 055, NASA Cooperative Agreement No. NCC9-16, Project No. IM.15, NASA RICIS Program, University of Houston, Clear Lake, Texas, U.S.A.

PRIOR, A., 1957, *Time and Modality* (Oxford: Oxford University Press).

SACERDOTI, E., 1977, *A Structure for Plans and Behavior* (New York, Elsevier Scientific Publications).

SCHWARTZ, S., 1977, *Naming, Necessity, and Natural Kinds* (Ithaca: Cornell University Press).

VENDLER, Z., 1967, *Linguistics in Philosophy* (Ithaca: Cornell University Press).

VENDLER, Z., 1968, *Adjectives and Nominalizations* (The Hague: Mouton, 1968).

ZACHMAN, J., 1986, A framework for information systems architecture. Report No. G30-2785, IBM Los Angeles Scientific Center, U.S.A.