4

A formal theory for knowledge-based product model representation

Filippo A. Salustri
The University of Windsor
Department of Industrial and Manufacturing Systems Engineering, The University of Windsor, Windsor, Ontario, N9B 3P4, Canada.
Telephone: (519) 253-4232 ext. 2621. Fax: (519) 973-7062.
e-mail: fil@server.uwindsor.ca

Abstract

The field of *design science* attempts to place engineering design on a more formal, rigorous footing. This paper introduces recent work by the author in this area. Artifact-Centered Modeling (ACM) is a general framework intended to partition the design endeavor in manageable sections. A fundamental part of ACM is the representation of information about products being designed. The Axiomatic Information Model for Design (AIM-D) is a formal theory about product information based on axiomatic set theory. AIM-D provides formal bases for quantities, features, parts and assemblies, systems, and subassemblies; these are all notions essential to design. It is not a product modeling system *per se*, but rather a logic of product structure whose axioms define criteria for determining the logical validity of product models. A previous version of the theory has been found to contain logical inconsistencies; the version presented herein addresses those problems. A complete axiomatization of the new theory is given, including a discussion of its validity. One of the obvious applications of AIM-D is in the development of knowledge-based systems for design. The author is currently implementing such a system using AIM-D as its foundation. The system, called *Designer*, provides the logical rigor of AIM-D within a computerized environment. At the user's level, the system appears to be an object-oriented knowledge-base capable of representing information about all the kinds of entities represented in AIM-D. Although still under development, a discussion of the design and implementation plans for *Designer* is given.

Keywords

design theory, set theory, knowledge-based system, product information model.

1 INTRODUCTION

It is generally acknowledged in both academia and industry, that engineering design as an endeavor does not stand on as formal a footing as the engineering sciences, though it is unclear why this should be

M. Mäntylä et al. (eds.), *Knowledge Intensive CAD*

so. The field of *design science* has emerged in response, to provide a more formal basis for design. The author's interest in this area is in the use *formal logic* to describe the structure of design so as to facilitate development of reasoning processes for, and about, design. Such work can obviously find application in the development of knowledge-based (KB) design systems. The author's current project is the development of (a) a formal theory of *product models*, and (b) to implement that theory in the form of a KB system for product modeling. It is important to distinguish the current work from many other efforts in product modeling. This theory does not provide a methodology for product modeling, but rather provides a logic of product structure. The axioms of the logic define criteria that must be met by any product model that is to be considered logically valid. This paper focuses on the author's theory of product models, and its implications for KB systems. The implementation phase of the project is on-going research and will be discussed here only in relatively general terms.

The current theory is a modified version of previously published work (Salustri and Venter, 1992), which (a) addresses a number of inconsistencies that have been detected since the original publication, and (b) expands the theory to include *systems* and *features*.

The premise of the author's research is that formal logic can be used to develop theories of design that are more robust than other, currently available theories, and that such theories will lead to representations and methodologies that will improve our collective ability to design high-quality, cost-effective products. The author has devised an overall framework to develop theories of the various aspects of the design endeavor; this framework will be presented briefly. While the overall work is not yet completed, this paper presents an important early result of the project: a theory of product structure derived from set theory. The theory is applicable in a number of areas, including the development of design *process* theories, curricula for teaching design, and computer-based tools to aid practicing designers. The focus in this paper is the development of computer-based tools.

The rest of this paper is organized as follows. First, a summary will be presented of the overall framework of the author's research. Next, the theory of product structure is presented, including its axiomatization. Then, a brief discussion of the validity of the theory is given, followed by a section discussing the use of the theory for the development of a KB system. After this, some extensions to the theory that are part of the author's current, on-going research will be briefly discussed. Finally, some related research of other workers will be examined, and a concluding section will summarize the author's findings.

2 OVERALL FRAMEWORK

The author has developed a general framework to partition the problem of describing design into manageable components. This framework, called Artifact-Centered Modeling (ACM), is shown graphically in Figure 1. ACM is developed from the premise that the engineered product should be at the heart of any modeling system intended to represent knowledge about design or designing. That is, a flexible, responsive enterprise should be able to tailor its organization to optimize its performance as well as the performance of its product(s); thus, the existing or anticipated product line is an essential input required to define the procedures and processes used by the enterprise.

ACM partitions the overall design endeavor by abstracting both by function and by structure (or representation). These abstractions form the axes of a two-dimensional matrix of design aspects. In Figure 1, the horizontal axis is for function abstraction. The most concrete level on this axis consists of the functions to be provided by the product (its functional requirements), and by the manufacturing processes required to fabricate it (see the top left of Figure 1). The first abstracted level contains the functions provided by whatever design process is in use; these functions take the product's functional

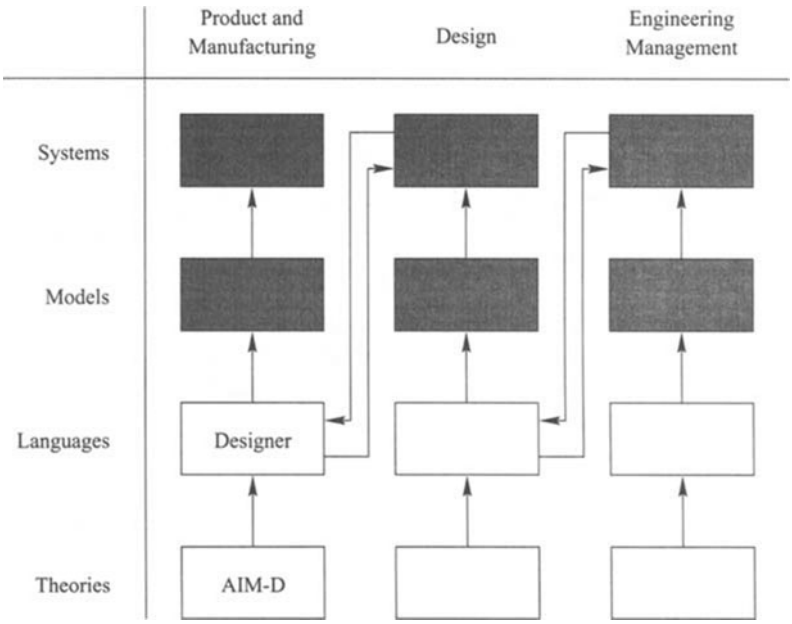


Figure 1 The artifact-centered modeling framework.

requirements as inputs and produce a product design as their output. At the next abstraction level are the engineering management functions that synthesize, monitor, and maintain the design process functions.

The other axis distinguishes four levels of representational abstraction. The most concrete level is that of systems: the product itself and the manufacturing equipment and processes needed to fabricate it. Once removed from this are models of the product and of the manufacturing system. At the next level of abstraction are languages that specify models (i.e. intentional descriptions of a set of possible models). Finally, at the most abstract level, are theories that justify languages used to model systems (i.e. intentional descriptions of sets of languages).

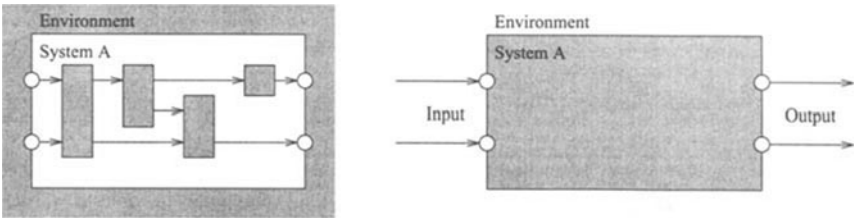
The two most concrete levels of representational abstraction (the shaded regions in Figure 1) are particular to each group of design agents. This group of agents may be engineers working on a particular product, or an entire design office working simultaneously on a number of different projects. Each group, whether they know it or not, have particular systems and models ranging from the product line through to the management systems. The remaining levels of representational abstraction fall largely into the realm of research; work at these levels is generally concerned with finding new, fundamental, robust, and efficient ways of treating products, design, and engineering management.

A variety of relationships exist between the elements of the ACM matrix, some of which are indicated by arrows in Figure 1. Obviously, theoretical developments affect the development of representational languages, which affect the kinds of models that can be developed with those languages and, in turn,

the designed products themselves. Similarly, engineering management systems affect design processes. But there are other relationships between ACM elements as well. For example, a design process affects the product model it produces. This is indicative of the effects that elements at one abstraction level have on elements at other abstraction levels. Also, the choice of language affects the design process that uses it. These relationships form a closed loop among the elements of the ACM matrix, which imply strong coupling between the various elements, and which make overall descriptions of design difficult to elucidate. However, the author believes that a systems approach, wherein individual elements may be treated as *black boxes*, can yield relevant and useful results. In particular, the author's work takes the following stance: it is possible to develop theories for each element of the ACM matrix as individual items, so long as a clear vision of the expected interrelationships between elements is maintained. This is the main contribution of ACM: it captures the major aspects of the design endeavor at a coarse level, yet can indicate clearly the interrelationships between them.

In addition to the decomposition of the elements of design by abstraction, ACM also assumes three *operational perspectives* of the elements represented in the ACM matrix. The *structural* perspective views the systems as *artifacts* produced by the design endeavor. The product itself is only one such artifact; in fact, every element of the matrix produces (possibly many) artifacts. The structural perspective deals with the representation of the *form* of those artifacts.

Secondly, there is a *behavioral* perspective. This perspective treats a system as a black box capable of translating a set of inputs into a set of outputs, where the *environment* in which the black box operates is "transparent". In this perspective, subsystems are synthesized into meaningful systems based on connecting outputs from one black box to inputs of another, independent of the internal workings of the subsystems. It is in this perspective that the interaction between a system and its general operating environment is defined.



(a) Example Functional Perspective

(b) Example Behavioral Perspective

Figure 2 Perspectives in ACM.

Finally, in the *functional* perspective, the system's environment is opaque, but its internal structure is transparent. In this perspective, the specifics of the transformation from inputs to outputs is described. The functional and behavioral perspectives are depicted graphically in Figure 2.

The distinction between behavioral and functional perspectives supports design processes that take advantage of *functional analysis*. At the initial stage of a new design, the behavioral perspective is used to study the environment and initial requirements of the design problem; these establish the overall inputs and outputs of the product to be designed. Once these inputs and outputs are established, the functional perspective is used to develop a transformation to produce the required outputs from the given inputs. The components of the system that provide the transformation are black boxes at this point. Note that

assuming a behavioral perspective from the point of view of an overall system (or product) is equivalent to assuming a functional perspective from the point of view of the elements of that system. By alternating behavioral and functional perspectives, a full functional breakdown of the design task can be synthesized. Further details regarding ACM can be found in Salustri (1995).

3 PRODUCT MODELING THEORY

Given ACM as described above, the next step is to begin filling in the details of the various elements of the matrix. The author's first step has been to focus on the lower left region of the matrix – specifically, the development of a logic of product structure.

A design's state is assumed to be representable at an instant by a collection of information. The theory's goal is to represent that information in a logically rigorous form. Also, a design process is considered as a sequence of actions leading from one logically valid state of information to another. If this is an acceptable view of design processes, then it is essential to have a solid understanding of the kinds of information present during a design process *before* the design process itself can be studied. This is the connection between the theory presented herein, about product structure, and design processes in general.

3.1 Background

The formal theory, called the Axiomatic Information Model for Design (AIM-D) is an interpretation of Zermelo-Fraenkel *axiomatic set theory* (commonly refered to as ZF) (Fraenkel et al., 1973). The goal of AIM-D is to provide a strictly logical framework for specifying information about a product at any point during its development. An interpretation of a ZF set theory is a mapping between the primitive entities in ZF and primitive entities in the domain of product structure. The mapping establishes a semantics for the axioms of ZF from a design standpoint.

There are two kinds of entities in ZF: individuals, and sets of individuals or of other sets. From the point of view of number theory, which has been classically the primary application of theories like ZF, it is possible to eliminate the individuals, thus allowing the hierarchy expressed by set membership to extend to both very large and very small sets. However, from a design perspective, this is not necessary: there will always be a point at which elements of a designed product can be considered primitive. So for AIM-D, we consider the primitive ZF entities to include both individuals and sets.

Also, the author has found that not all the axioms of ZF are needed to describe product information formally. The axioms that have been found useful to develop AIM-D are those of identity, separation, foundation, power set, and union.

3.2 Problems with the original theory

Since its original publication, the author has identified some logical inconsistencies in AIM-D. Firstly, it was possible to represent an assembly, A, that has a part, P, such that both assembly and part had an identical characteristic, such as overall length, l. Using the standard notation of set theory, this situation can be represented with the three statements: $P \in A, l \in P, l \in A$. However, these statements violate the ZF Axiom of Foundation, which states that no member of one set that is also a set may contain another member of the one. This means that the situation above, natural and intuitive though it may seem, is logically inconsistent.

Secondly, some statements in the original theory were *second-order* in nature. For example, the use of explicit domains, which captured type information, took the form of statements about terms in the theory, rather than statements about the real objects those terms represented. Although there is a growing interest in second-order logics (Shapiro, 1991), first-order logics are far better understood, and have found to be largely useful in knowledge representation, artificial intelligence, mathematics, and computer science. Thus, it would be preferable to be AIM-D as a first-order logic.

Thirdly, some kinds of entities clearly relevant to modeling geometry were difficult to represent. For example, if a spatial vector was defined as an entity, it could be used as a *component* of another entity, but not as one of its *properties*. But normal vectors of planes are better described as properties, rather than components, of the planes. Should vectors then be property values, or entities?

These problems made it difficult to construct KB tools for design based on AIM-D, and needed to be addressed. A new development of AIM-D is given below that addresses all of these problems.

3.3 Quantities

ZF individuals are mapped to the most fundamental units of information that can describe quantitative information about products. In AIM-D, these units are called *quantities* and are tuples of numeric values and dimensional metrics; 5 feet, 200 MPa, and 10 ohms are all examples of AIM-D quantities. It is essential that dimensional information be embedded in the theory since without such information, quantitative entities are not logically comparable; that is, quantities must be *ordinal* values.

We assume that a product model, **M**, contains a finite and countable number of quantities. This is a modest assumption, met by any realistic model, but that cannot be proved logically. Quantities map to ZF individuals, and any set composed only of individuals is a valid set since a set not containing other sets cannot violate the axioms. We can therefore define the following:

Definition 1 *All the quantities in a model,* **M**, *can be gathered into a set, called* **Q**.

Furthermore, the collection of values of all quantities, and of the dimensional metrics of all quantities, are each assumed to be finite and countable, and may be gathered into sets:

Definition 2 *The collection of values in a model* **M** *form a set, named* **V**.

Definition 3 *The collection of dimensional metrics in a model* **M** *form a set, named* **D**.

This permits quantities to be defined formally as follows:

Definition 4 *A quantity, q, is a tuple and an element of the cartesian product of the sets* **V** *and* **D**.

$$\forall q[(q \in Q) \rightarrow \exists v \exists d[(v \in \mathbf{V}) \bullet (d \in \mathbf{D}) \bullet (q = \langle v, d \rangle)]] \tag{1}$$

A function of convenience is also defined, which maps quantities to dimensional metrics: for a quantity q, $\text{metric}(q)$ is such that $q = \langle v, \text{metric}(q) \rangle$.

Finally, a set of primitive metrics is identified, from which all other metrics can be derived, according to the standard rules of dimensional analysis. In AIM-D, the following metrics are considered primitive:

- length (including angular displacement)
- time
- voltage
- cost
- NDU (non-dimensional unit – for ratios, etc.)
- mass
- temperature
- amperage
- enumeration (for counting items)

This list includes metrics that are not typically considered in the physical and engineering sciences. Metrics are *rational* values invariant over any model of any product (e.g. a length is always a length). This fact will be used to develop the axioms of *entity type*, below.

3.4 Design entities

Individuals are collected into sets, sets of sets, etc. In AIM-D, these sets are called *design entities* (DEs). A product model is a DE that may be decomposed into other DEs until the level of quantities is reached.

However, there are various levels between DEs and quantities where important kinds of non-primitive information arise. A product structure theory must treat these other information units as well. In AIM-D, there are four kinds of information: quantities, features, parts, and assemblies. Quantities have been discussed above; the other three levels are discussed below, and are all represented by DEs.

Identity of DEs is defined by the straightforward application of the ZF Axiom of Identity. This yields:

Axiom 1 (Identity of design entities) *Two design entities, X and Y, are identical if they share exactly the same elements.*

$$(X = Y) =_{df} \forall x[(x \in X) \equiv (x \in Y)] \tag{2}$$

DEs can have a variety of kinds of elements. One kind of DE element is a *property*, which is an intrinsic characteristic of the DE. Most logics, and many KB systems and AI frameworks, take advantage of this notion. But, surprisingly, there is no clear, universal definition of what a property is exactly (van Dalen, et al., 1978). In our case, this is actually advantageous, since it allows us to define the concept without risk of violating the rules of logic. In AIM-D, it is sufficient to define a property as an intrinsic characteristic of a DE. Properties may have values that are either *explicit – 5 feet, through-hole, spur gear*, and *stainless steel* are examples of property values – or *derived*, in which case the value of the property is calculated based on other properties – for example, physical volume calculated from dimensions and shape.

DEs are partitioned into sub-entities by various criteria that selectively include only certain elements. For example, given a product model, extracting information about shape yields a geometry model of the product. In AIM-D, this partitioning is achieved with *views*. A view is a *subset* of the elements of a DE, plus relations between those elements. The ZF Axiom of Separation is used to define views formally.

Axiom 2 (Axiom of view formation) *A DE, Y, that is a view of another DE, X, consists of all elements of X that satisfy a predicate γ.*

$$\forall X \qquad \exists Y[\forall x[(x \in Y) \equiv (x \in X) \bullet \gamma(x)]] \tag{3}$$

The predicate γ is true for all DE elements that are in the relation that describes a view. For example, given a DE representing a four-bar linkage, and four instances of a relation *freely-pinned*, γ would be

true only for the parts entering into the *freely-pinned* relations. The resulting view would contain the mechanical elements that are freely pinned to each other. Since Axiom 2 determines logically valid views, it also restricts the logically valid relations. The relations that can be used in Axiom 2 are essential aspects of AIM-D. They may be considered constraints on the behavior of two or more DEs, or functions provided by the system modeled by the view and that cannot be provided by the elements of the view individually. It is unclear which perspective yields a more coherent theory at this time, but it is clear that both perspectives are meaningful and will have to be addressed.

Views are not combined using set membership in AIM-D. That is, a view cannot have other views as members. Combining views is done with a set-wise union operation. Thus the union of all views of a product model is the product model itself. The ZF Union Axiom is used here.

Axiom 3 (Axiom of view combination) *A DE, X, is a composition of all the views containing elements of X.*

$$\forall X[\exists V[\forall x[(x \in X) \equiv \exists v[(x \in v) \bullet (v \in V)]]] \tag{4}$$

where X and x are DEs, v is a view, and V is a set of views.

It is noted that valid views are subject to the Axioms of Separation and Foundation. Also, a given DE may occur in many views in V and still occur only once in X; duplicate instances of a DE are identified through Axiom 1.

3.5 Features

The primitive *physical* entities in a product are the "atomic" parts that are assembled to form it. Castings, moldings, and machined items are all parts. However, part geometry information is needed when defining the exact physical relationships between them. The components of part geometry are regarded as *features*. It is evident from the literature that a universally accepted definition "features" does not exist; for example, the definitions used by Pabon et al. (1992), Rosen (1993), and Henderson (1993) are significantly different from each other. In AIM-D, features describe the geometric aspects of parts that enter into physical relationships. For example, information about the sides/edges of two plates is needed when those plates are to be welded together; the sides/edges are the anchor features at which the weld relation is associated. Features are distinct from parts in AIM-D in that features are not *realizable* (manufacturable), except as elements of parts. Features are the fundamental information units in AIM-D to which physical relationships are associated. Features may be values of part properties. It is noted that features are considered here only from a design perspective, although the manufacturing aspects of features are also very important.

We assume there are a finite and countable number of features in a product model.

Definition 5 *The features of a model* **M** *compose a set called* **F**.

Features are formalized, using the ZF Axiom of Separation, as sets of quantities plus relations between the quantities.

Axiom 4 (Axiom of feature formation) *A feature, f, is a DE that is composed of a set of quantities, each element of which satisfies a predicate α.*

$$\exists f[\forall q[(q \in f) \equiv (q \in \mathbf{Q}) \bullet \alpha(q)]] \tag{5}$$

3.6 Parts and assemblies

Parts are the primitive physical components in a product model. Though the value of one part property may depend on other property values (e.g. weight may depend on density and volume), they never depend on the values of properties of *other* parts. Parts are thus treated as DEs in AIM-D.

Definition 6 *The parts of a model* **M** *compose a set called* **P**.

An axiom for the treatment of parts as combinations of features may be written as follows.

Axiom 5 (Axiom of part formation) *A part, p, consists of all features, f, that are in F, and that satisfy a predicate ϕ.*

$$\exists p[\forall f[(f \in p) \equiv (f \in \mathbf{F}) \bullet \phi(f)]] \tag{6}$$

Parts are gathered into *assemblies*. Assemblies also have properties (length, mass, etc.). All assembly properties are derived from the properties of the assembly's parts and are not intrinsic to the assembly itself. For example, though an automobile may be thought of as having mass, it is really the parts of the automobile that have mass. The automobile's mass can be changed by changing parts, but the mass of a part is constant (assuming the part is not machined or otherwise altered in a way that alters its mass – in which case, one may argue that the original part not longer exists).

The part/whole relationship is basic to how humans regard systems, natural or artificial, designed or otherwise. *Mereology* is the branch of logic that deals with this relationship. Mereologically, there is more to an assembly than just a collection of parts. That is, it is insufficient to simply lay out all the parts of an automobile on the garage floor to have an automobile; the parts must exist in *definite relations*.

In summary, an assembly (or part) consists of a set of parts (features), plus relations that exist between parts (features). This definition is consistent wiht that of a *graph*. The author expects that graph theory will play a significant role in the future extensions of AIM-D.

Another mereological concept, that of *subassemblies*, introduces interesting complexities. The concept is relative: a subassembly may be treated as *a part of an assembly*, or as *an assembly of parts*. Its true usefulness in design is to form meaningful part aggregations that eliminate detail about a its components while preserving the essential characteristics of the aggregate itself. In other words, subassemblies are *abstractions* of collections of parts, useful as place-holders for information that is indeterminate in the early stages of a design process. Subassemblies are also very important in manufacturing and assembly.

Difficulties arise when considering relations *between* subassemblies. To say, for example, that *an automobile's engine is connected to its chassis*, is true but vague. More precisely, we could say that *there are pickup points on the engine block that are bolted to matching items on the chassis*. The vaguer phrasing is sufficient because the human mind can extract implicit information from it easily. But to support computer-aided reasoning with this kind of information, there must exist the means to capture the implied information explicitly. It is for this reason that we concern ourselves with these details here.

It is evident from this example that physical relations exist at the level of *parts* (or even part features) of subassemblies. Subassemblies may connect in multiple ways to achieve multiple functions. While sufficient information may be available to clearly define relations in structural terms late in a design, it is the *goal* of design to reach that stage. Since subassemblies are abstractions of part collections, the relations between subassemblies are abstractions of the relations between the assemblies' parts. Since the *form* of the parts entering into a relation is unknown, the abstraction must depend on *function*. So relations between subassemblies will usually contain functional, rather than structural, information.

This discussion motivates the perspective taken in AIM-D regarding subassemblies: while they are essential design concepts, they are *not* fundamental for logical product representations. Parts are the atomic components of a product, and assemblies represent the total product. A subassembly can be treated as a *subset* of the set of parts, plus the necessary relations. This allows the application of ZF to establish rules about what constitutes a logically valid subassembly.

In fact, subassemblies are a kind of *view* in AIM-D. So, rather than treating subassemblies as parts that retain their status as parts when they are combined into an assembly, AIM-D requires that the subassemblies be broken open in the assembly, just as views are treated. The subassemblies may still exist as separate DEs, and may refer to the same parts as the assembly, but only the parts of the subassemblies (and the relations between them) are contained in the assembly. This leads to the following axioms.

Axiom 6 (Axiom of subassembly formation) *A subassembly, S, consists of all parts, p, that are in* $\mathbf{P}$*, and that satisfy a predicate δ.*

$$\exists S[\forall p[(p \in S) \equiv (p \in \mathbf{P}) \bullet \delta(p)]] \tag{7}$$

The predicate δ is defined by the relation for a given subassembly. For example, in a freely-pinned connection between two parts, δ specifies the parts (and fastener) that occur in the relation.

Axiom 7 (Axiom of subassembly combination) *For all assemblies, A, there exists a set of subassemblies, S, such that at every part in A occurs in at least one subassembly, s, in S.*

$$\forall A[\exists X[\forall p[(p \in A) \equiv \exists s[(p \in s) \bullet (s \in S)]]]] \tag{8}$$

The notation $\cup S$ denotes the union of a set of subassemblies S.

3.7 Systems

There is another useful mereological structure: a *system*. Systems-based design is a popular paradigm, especially useful in the early stages of a design process, because a system, like an AIM-D sub-assembly, is an abstraction of a collection of functions to be provided by a product. In fact, a sub-assembly and a system are roughly equivalent. The definition given in Karnopp et al. (1990), which characterizes systems are (a) definitely distinguishable from their operating environment, and (b) composed of interacting parts, admits both sub-assemblies and systems. Where system components are connected functionally, subassembly parts are connected physically; so systems are actually more general. This means that a system is a collection of DEs that enter into some relation. This is exactly what an AIM-D *view* is; indeed, in AIM-D, the notions of systems and views are interchangeable. Thus the axioms that treat views also provide the required formalization for systems.

3.8 Type information

The preceding section discussed mereological modeling in AIM-D. There is also, however, a *taxonomic* aspect that must be treated. A *taxonomy* is a structure used to classify items according to some set of criteria. Taxonomic information allows individual components to be identified, compared, and, often, referred to. The names of concepts for classes of things are often used to name individual members of those classes. For example, the sentence *"The central gear is fixed to the axle"* may refer to a specific individual gear and axle in a particular mechanism even though it uses the general concept names of *gear* and *axle*. Because this is obviously a very natural way to express design information, and because this kind of conceptualization forms the basis for a system of *types* of design information, it is very important to establish rigorous conventions for these concepts so that they may be treated logically.

The mind maps real entities to conceptual names through *generalization*, which may be defined as selectively neglecting known dissimilar aspects of items in order to discover otherwise hidden or unknown similarities. This kind of information allows the *intentional* definition of entities, and is particularly useful to classify items based on conceptual notions that may change with time. Intentional information is treated directly by logics called *description* or *term* logics; these logics have provided the foundations for a number of KB systems that have found application in engineering, such as KIF (Hakim and Garret, 1993) and CLASSIC (Brachman et al., 1991). This is one of the major advantages of KB technologies; intentional information is often unavailable, or at least difficult to provide and access, in other data modeling schemes. In order to formalize generalization, we must determine what kinds of information may be neglected. The author identifies three categories of information generalization: ignoring values of properties or parts, ignoring entire properties and parts, and ignoring the number of property or part values. Each of these is examined below.

Information may be generalized by ignoring the values of properties and parts of a DE. The remaining information will only assert whether a DE has a given property or part. For example, we can assert that an automobile has an engine, but not what the engine actually is. This kind of generalization captures the notion of *type*, and is common in programming languages, information modeling systems, and KB and database systems. In this sense, two DEs are of the same type if corresponding pairs of attributes are of the same type. This is a recursive definition that ends at the level of quantities. So in order to axiomatize this kind of generalization, we must examine type-compatibility of quantities.

In AIM-D, two quantities are type-compatible if they have the same dimensional metric, and two DEs are type-compatible if corresponding pairs of elements, one from each DE, are type-compatible. The predicate *isa* is defined to capture this notion:

Definition 7 (The type-compatibility predicate)

$$\text{isa}(q,r) \quad =_{df} \quad (q \in \mathbf{Q}) \bullet (r \in \mathbf{Q}) \bullet (\text{metric}(q) = \text{metric}(r)) \tag{9}$$

$$\text{isa}(x,y) \quad =_{df} \quad \forall a[(a \in x) \bullet !\exists b[(b \in y) \bullet \text{isa}(a,b)]] \tag{10}$$

where the notation $!\exists$ *is read* "...there exists exactly one..." *(Bernays, 1968).*

By dealing with DEs in general, *isa* applies equally to features, parts, assemblies, subassemblies and systems.

The set of parts, **P**, and the set of features, **F**, each can be used as the basis on which type-compatibility can be defined with the isa predicate. Using the Axiom of Separation leads to the following.

Axiom 8 (First axiom of generalization) *Each part p in* **P**, *and each feature f in* **F**, *is exemplary of the members of a set S, all the members of which are type-compatible with p or f with respect to* isa.

$$\forall p \quad \exists S[\forall x[(x \in S) \equiv (x \in \mathbf{P}) \bullet \text{isa}(x, p)]] \tag{11}$$

$$\forall f \quad \exists S[\forall x[(x \in S) \equiv (x \in \mathbf{P}) \bullet \text{isa}(x, f)]] \tag{12}$$

where x is an exemplar representative of a number of other parts in the model; in other words, x is an exemplar of a type of part.

This approach does not require explicit types and is thus first-order; it can accomodate a virtually unlimited number of simultaneous type hierarchies over the same set of parts, and requires no bookkeepping overhead associated with consistency maintenance of both types and instances.

The second form of generalization involves the dismissal of whole properties. For example, a window may be defined as an item composed of a material that is transparent; all other properties are irrelevant to identifying windows. If a DE has a *set* of properties and a *set* of parts, then this kind of generalization involves identifying meaningful *subsets* of those sets. ZF's definition of a subset can be applied directly here: any subset that satisfies the ZF Axiom of Separation represents a logically meaningful generalization of DE. Thus, each subset of a part is a generalization of that part, and thus is an exemplar of a *super*type of the part. Furthermore, the power set of a part p, $\mathcal{P}(x)$, contains all the possible supertype exemplars of p, and a set, **S**, can be defined to contain all the supertype exemplars of all parts in a product. Similarly, a set **T** of all possible supertypes of all features in a product can be defined also.

Definition 8 (All Possible Supertype Exemplars of a Model)

$$\exists \mathbf{S}[\forall s[(s \in \mathbf{S}) \equiv \exists p[(p \in \mathbf{P}) \bullet (s \in \mathcal{P}(p))]]] \tag{13}$$

$$\exists \mathbf{T}[\forall t[(t \in \mathbf{T}) \equiv \exists f[(f \in \mathbf{F}) \bullet (t \in \mathcal{P}(f))]]] \tag{14}$$

A new predicate is needed to relate supertype exemplars to parts. The predicate *specializes*, defined below, fills this role, and differs from isa only in the use of $\exists$ instead of $!\exists$.

Definition 9 (The Specialization Predicate)

$$\text{specializes}(q, r) =_{df} (q \in \mathbf{Q}) \bullet (r \in \mathbf{Q}) \bullet (\text{metric}(q) = \text{metric}(r)) \tag{15}$$

$$\text{specializes}(x, y) =_{df} \forall a[(a \in x) \bullet \exists b[(b \in y) \bullet \text{specializes}(a, b)]] \tag{16}$$

Finally, the Axiom of Separation defines a generalization axiom relating supertype exemplars to parts.

Axiom 9 (Second axiom of generalization) *A supertype exemplar of a part (or a feature) of a model is a generalization of a set of parts (or features) in the model.*

$$\forall s \quad \exists S[\forall x[(x \in S) \equiv (x \in \mathbf{P}) \bullet \text{specializes}(x, s)]] \tag{17}$$

$$\forall t \quad \exists S[\forall x[(x \in S) \equiv (x \in \mathbf{F}) \bullet \text{specializes}(x, t)]] \tag{18}$$

where s is an element of **S**, *and t is an element of bf T, as defined above.*

It is also possible to define the inverse of specializes very simply as: generalizes(s, x) = specializes(x, s).

Not all subsets derived from the axioms above will be meaningful from an engineering perspective. That is, AIM-D is not used directly to generate reasonable generalized DEs of a given product; rather, it is a valuable tool to ensure that the DEs that are identified by a designer satisfy logical criteria.

The third kind of generalization involves ignoring the number of values of a property, part, or feature. This is AIM-D's most significant departure from techniques often endorsed by the object-oriented technology community. In an object-oriented environment, one might model an automobile with an object having an attribute for the *number* of wheels, and another attribute for the *kind* of wheels are to be used in a particular automobile. AIM-D, on the other hand, adopts the perspective taken by *description logics* used in knowledge representation, such as that proposed by Nebel (1990), wherein an automobile is be modeled as having wheels, and placing restrictions on the number and type of wheel "instances" in a particular automobile. The author prefers the description logic approach because it does not require attributes that have no corresponding manifestation in the real-world item being modeled (i.e. an attribute indicating the number of wheels). Rather, it allows values of properties, parts, and features to be grouped so as to make the same information available. In the sense that the resulting model corresponds more closely to the actual object being modeled, this approach is seen by the author as being more *natural.*

A property, part, or feature of a DE may have a *set* of values, and the cardinality (size) of that set may be ignored or limited in order to generalize the DE. This permits a variable number of values for a property or part of a DE. For example, automobile engines can be characterized as having *between* 2 and 4 valves per cylinder. In order to treat this kind of generalization, the number of elements in a set must be determinable: this is given by card(A), where A is a set. The predicate *within* is defined in AIM-D to determine if the cardinality of values of a property, part, or feature falls within a given range.

Definition 10 (The cardinality predicate) *For a supertype exemplar g and two integers i and j, the predicate* within *is true if the cardinality of all the attributes of a part p that are common to g have sets of values whose cardinality is between i and j.*

$$\text{within}(g, p, i, j) =_{df} \forall a[(a \in g) \bullet \exists b[(b \in p) \bullet i \leq \text{card}(b) \leq j]] \tag{19}$$

This leads to the following interpretation of the Axiom of Separation for cardinality generalization.

Axiom 10 (Third axiom of generalization) *A part p or a feature f is an exemplar of a set of parts or features, each element of which has attribute value sets whose cardinality corresponds to the cardinality of the exemplar.*

$$\forall g \forall i \forall j \qquad \exists S[(p \in S) \equiv (p \in \mathbf{P}) \bullet \text{within}(g, p, i, j)] \tag{20}$$

$$\forall g \forall i \forall j \qquad \exists S[(f \in S) \equiv (f \in \mathbf{F}) \bullet \text{within}(g, f, i, j)] \tag{21}$$

This completes the development of the axiomatic foundation of product model information with AIM-D. The theory integrates notions of quantities, parts, features, and assemblies, as well as subassemblies and systems, into a consistent system of logic for product models.

4 VALIDITY OF AIM-D

Validity is a primary concern in all logical theories. A valid theory is one in which all true statements can be proved true within the system. While Goedel's Incompleteness Theorem casts doubt on the validity of set theory and other useful systems of logic (Hofstadter, 1979), it *is* possible to show relative validity between systems. Specifically, there is a theorem of logic that proves that any extension of a system that introduces no new quantifiers, primitives, or connectives, is valid with respect to the original system (Copi, 1979). AIM-D is an extension of ZF set theory that obeys this restriction. Therefore, AIM-D is *as valid a system of logic as set theory itself.* This is a fundamental result of the author's work because it sets the foundations for structured and even automated reasoning about designed products with a degree of rigor unique in all the design research of which the author is aware.

This is not to say, however, that any product model that satisfies the axioms of AIM-D is necessarily superior; there is no assurance that a model consistent with AIM-D will cost less, or be of higher quality, than other models. But this is not AIM-D's purpose. Rather, AIM-D is intended to (a) identify logical inconsistencies in models, on the premise that improving the logical rigor of a model will improve its overall adequacy, and (b) specify information about product so as to allow their quantitative evaluation and comparison. This requires, in turn, that sufficient information about a design be represented (including cost, quality, etc.). Such models can then be analysed in a logically rigorous manner to determine their viability when compared to other existing models.

5 APPLICATION OF AIM-D TO KNOWLEDGE-BASED SYSTEMS

A detailed axiomatization of product structure based on set theory has been presented. The result, AIM-D, is valid with respect to its axiomatic foundation. An important application for AIM-D is in the development of KB systems for design. This section will discuss the author's work in this area.

Clearly, AIM-D's notion of sets of entities can be treated computationally by an object-oriented technology. However, there are two significant differences between the author's approach and that of conventional object-orientation. Firstly, type information in AIM-D is not explicit, which ensures that AIM-D remains a first-order theory. Most object systems, on the other hand, are *class-based,* having explicit structures to capture type information (for example, C++ or CLOS), and may contain second-order information. This suggests that a computational system based on AIM-D should be based on *prototypes* rather than classes. Prototype-based systems have distinct advantages over class-based systems (Johnson and Zweig, 1991), especially in engineering applications; unfortunately, there are very few such systems, none of them having reached the maturity expected of a commercial product. Secondly, there are kinds of type information – such as classification based on the number of values of an attribute – that are treated differently in AIM-D. The author has attempted to reconcile these differences between AIM-D and object-orientation. The resulting system, called *Designer*, is implemented in the Scheme programming language (IEEE, 1991). *Designer* is intended to represent the state of product information at any point in its development, while satisfying the axioms of AIM-D. The language appears as a prototype-based, object-oriented programming language with limited multiple inheritance, generic functions, and a LISP-like syntax. *Designer* has been discussed in detail in other publications (Salustri and Venter, 1994, and Salustri, 1996).

Although the author has had some success with the object-oriented approach, the resulting implementation of *Designer* has been cumbersome to implement robustly, slow, and difficult to use. The author believes this is a result of the shortcomings mentioned above.

Frame-based systems (Gonzalez and Dankel, 1993), on the other hand, exhibit characteristics that make

them more suitable as an implementation platform for AIM-D. Frames support prototypes, multi-valued attributes, and various operations on those structures, in a more flexible manner than typical object-oriented methodologies. They are commonly used in KB systems. Therefore, the author has decided to re-implement *Designer* taking advantage of frames, rather than objects. Scheme is still used as the implementation language, primarily for its succinct syntax, and the ease with which systems can be quickly prototyped. The new system is still under development, but some remarks regarding its features and capabilities are in order here.

Frames in *Designer* model AIM-D design entities. They are also used for other implementation features, such as generic functions, but this aspect is hidden from the user's view. AIM-D quantities are represented as separate data items.

One essential difference between AIM-D and its implementation in *Designer* is that while the theory deals with the structures used to model products, its implementation must treat the *naming* of those structures. The issue would be relatively simple if *Designer* were intended for use by individual designers; but instead, *Designer* is intended to be used in collaborative environments including many designers. The issue of how the same things are named (identified) by different agents is an open problem in knowledge representation research, and there are a number of efforts intent on discovering its underlying logic (e.g. Capoyleas et al., 1996, Buvac et al., 1995, and Grove, 1995). The approach taken in *Designer* is fairly common in other KB systems: a *context* is defined as a mapping frame names to the frames themselves. Contexts may be nested; the outermost context contains various system definitions and represents a name-space containing universal concepts, and successive inner contexts become more specific to individual tasks and designers/users. Names can be reused in different contexts to denote different frames; this allows, for example, different users to name the same frame differently (or different frames the same). Although the problem of managing contexts effectively has not been dealt with yet by the author, the basic functionality as described here is well-defined as of this writing.

All frames have explicit names in at least one context; there is a mechanism to automatically generate unique but meaningful names for frames created without an explicitly given name. Frames can be referred to by their explicit names, or by an indirect name composed of a *path* leading from some frame to the frame in question. For example, let `car#22` be the explicit name of a frame representing a particular automobile, and let `v8-3.5l#127` be the explicit name of a frame representing the engine of `car#22`. The term `(of engine car#22)` is a path leading to the same frame as the explicit term `v8-3.5l#127`.

Relations are also represented by frames. For example, the definition of the term engine can describe it as a relation between car frames and engine frames; it describes the relation's domain and range in terms of exemplars, according to the type axioms in AIM-D. The definition of engine can also describe it as a prototypical automobile engine. The definition used in each occurence of the term depends on the context that is current at the time the term is evaluated, and on the role assumed by the term in a statement. Thus, in the example above, engine occurs in the role of the name of an attribute of `car#22`; on the other hand, the role played by engine in a form like *(of piston engine)* is that of an exemplar DE.

Although frames are used to represent quantities (**Q**), features (**F**), and parts (**P**) in a model **M**, it is not possible to enumerate all defined frames. However, it is possible to enumerate all members of each of the sets **Q**, **F**, and **P**, in a model; this allows the construction of algorithms that operate, for example, over all parts in a product model. It follows from AIM-D that a quantity cannot contain another quantity, a feature cannot contain another feature, and a part cannot contain another part. Similarly, features are composed only of quantities, parts may be composed of features and quantities, and assemblies may be composed of parts and quantities. These rules are all enforced by *Designer*.

Designer stores type information by maintaining a specialization hierarchy. Internally, this is done by using a special relation attribute, `specializes`, whose value is a list of frames that specialize a given frame. The inverse relation, `generalizes`, is also maintained. The AIM-D type predicates *isa*, *specializes*,

and *within* are all implemented as well. These attributes and predicates are used in situations where a user wishes to define an explicit specialization relationship. In addition to these explicit type-checking capabilities, *Designer* also allows type-checking by structural comparison of frames. For example, a frame A is a specialization of a frame B, if all attributes in B have type-compatible counterparts in A; this procedure is recursive, ending at the level of quantities. These capabilities allow the system to find specialization relaionships that are not explicitly given by the user, and to verify that any explicitly given specialization relationships are consistent with the descriptions of the frames involved.

The notions of views, systems, and subassemblies in AIM-D are all represented with DEs that are treated as subsets of other DEs. Views do not contain other views, and likewise for systems and subassemblies. There are exemplar frames for each of views, systems, and subassemblies, that are the roots of specialization trees; in this way, these kinds of entities are distinguishable from quantities, parts, and features. The algorithms used to create new frames make use of this distinction to ensure AIM-D's rules regarding views, systems, and subassmblies are obeyed.

This discussion outlines some of the major features of the new version of *Designer*. Once completed, the new version, like its predecessors, will also include a library of generally useful frames representing common generic entities. From these generic libraries, other modules will be developed that will be particular to specific domains (e.g. mechanisms, pumps, motors, etc.).

Designer clearly has limitations. For example, there is currently no way to associate a function with a part or assembly (i.e. no way to represent the fact that a certain part or assembly provides a certain function). There is no support for time dependencies, or for qualitative information (commonly found in the early stages of a design process). This is because there is as yet no logical infrastructure in AIM-D to support it. The language will grow as new theoretical developments are added.

6 EXTENSIONS TO THE BASIC THEORY

Although AIM-D fully describes the representation of structure in product models, there are a number of avenues that may be pursued to extend its capabilities even further. This section briefly discusses two areas that are of current interest to the author.

6.1 Representation of function

It is clear from even a cursory inspection of the current literature that the cost and quality of designs is most sensitive to changes made in the earliest phases of a design process, namely in *conceptual* and *embodiment* design. The author believes that these phases have been largely resistant to attempts at formalization, due in part to a relatively poor understanding of the *function* of products, and to how function maps to form. Although a number of research efforts exist – for example, IDEF-0 (NIST, 1993), and bond graph theory (Karnopp et al., 1990) – none has met with significant acceptance. This may be due to the somewhat arcane nature of the systems, or to the relatively restricted domains over which they operate. Nonetheless, this area continues to be investigated vigorously.

If AIM-D is to be a universal logic of product models, it *must* treat functions of products and how those functions relate to the form of the artifact. The most primitive and abstract functions in electromechanical design are transfers of energy, of information, and of material (or mass). These are disjoint functions, and may be gathered into a set. A transfer occurs *between* two (or more) items, and thus is directed. The direction, location, and magnitude of these transfers are their basic properties. From the set

of primitive functions, and the ZF Power Set Axiom, and from relations between functions and entities that either provide, or are acted upon by, those functions, a hierarchy of more complex functions can be constructed. The author believes an axiomatization of function is possible, similar to the axiomatization of structure given in this paper.

The basic transfer functions noted above are fundamental to design. But an artifact must provide other functions as well, functions relating to the needs and preferences of the customer or user, to maintainability, assembly, and manufacturing, to the corporate aims of the engineering enterprise, and to the environment in which the product is to be used. These kinds of functions have only recently attracted the attention of researchers, and much work remains to be done in this area. Although the author has not yet investigated the issue of function modeling to any depth, it remains a high priority for future research.

6.2 Testing and analysis

The axioms of AIM-D may be used as rules of logical integrity of information, from which algorithms to check information stored in a computer may be constructed. Other rules may be derived to automatically perform certain actions, such as classify DEs. Three examples of rules deriving from the axioms are:

1. No entity may contain both features and parts; if this were possible, then the ZF Axiom of Foundation would be violated (in the general case), resulting in an inconsistent product model.
2. A subassembly is identified by a relation. Without that relation, there is no way to distinguish between elements of the subassembly, and other DEs. Therefore, no subassembly may be formed without *first* defining an appropriate relation.
3. The axioms of generalization are classification criteria that can partition a set of DEs into subsets based on similarities between corresponding elements from pairs of DEs. This means it is possible to automatically classify any DE and, for instance, insert a new DE into a hierarchy of existing DEs.

Additionally, other validity checks may be performed by using the axioms as the basis for inference algorithms. *Deduction* is the classical inference technique of logic, and may be used with AIM-D's axioms as well. The rules of deductive inference allow a variety of *theorems* (statements whose truth is not known) to be either verified or denied through application of the axioms. However, it is evident that deduction alone is not enough to support the myriad actions that may be performed on product models in the course of a design process. There are occasions when *induction* or *abduction* may best represent the actions being performed.

As AIM-D matures, the author will begin to investigate how the various inference techniques can be applied best to produce algorithms to (semi-)automatically check the validity of product models. Such verifications could significantly improve a designer's ability to *justify* a design on logical grounds.

7 RELATED RESEARCH

It appears that the original impetus to treat aspects of design logically arose from the need to provide computer tools to aid the practicing designer. Theories of geometric modeling were developed to support computer-based graphics and, eventually, solid modeling systems. In the end, the theory of geometric modeling became its own field of study. Similarly, with the realization that artificial intelligence research

could play an important role in engineering, researchers began to regard logic as a means of formalizing engineering knowledge. It is not unexpected, then, to find that much of the research most closely related to AIM-D comes from the computational fields.

Description logics, such as those underlying KIF and CLASSIC, have certain points of interest in common with AIM-D; this has already been discussed. However, these logics are targetted to the general problems of knowledge representation, and, unlike AIM-D, are not intended to treat the particular requirements of engineering. Logical foundations are also present in a number of other computational systems. For example, the EDM data model of Eastman et al. (1991) is directed to the construction of databases for product modeling using logic; SUMM (Fulton, 1992) attempts to provide an underlying logic based on the predicate calculus for the PDES/STEP product model initiative; and the logic-based expert system developed by Jaluria et al. (1991) is directed at a particular design task. None of these efforts fill the role that AIM-D is designed to fill, namely, to provide a foundational understanding of the product itself, rather than *models* of those products.

Logic has also been applied to the development of other abstract systems that are relevant in engineering. Examples include the development of constraint theory (Friedman and Leondes, 1969), the topological aspects of feature-based design (Rosen and Peters, 1992), systems for cooperative design (DePaoli and Tisato, 1994, and Polat et al., 1993), and language-based systems for design (Ward, 1992). These efforts, though relying on logic to an extent, do not preserve the degree of rigor afforded by AIM-D; their validity is therefore suspect (from a logical perspective).

Finally, logic has been and continues to be used as a fundamental means of studying design in a purely abstract sense. Examples of this include Bijl (1987), Suh (1990), and Roozenburg (1992). These efforts, among others, focus primarily on the *processes* of design, rather than on the formalization of the informational content associated with those processes.

In light of the forgoing review of related research, the author suggests that AIM-D straddles a number of different existing design research arenas. It is well-suited to computational developments; this has been a key focus point. But it also preserves a level of abstraction suited to academic research, and though it focuses on product information, it may also find use as a foundation upon which to build process theories.

8 CONCLUSIONS

This paper has introduced AIM-D, the Axiomatic Information Theory for Design, a theory able to describe the structure of designed products in a logically rigorous manner. It is not a product modeling system in itself, but rather a logic of product structure whose axioms define criteria to determine the logical validity of any product model. The previous version of AIM-D exhibited certain logical difficulties that have been resolved in the version presented here. AIM-D is based on axiomatic set theory, and is demonstrably valid with respect to set theory. It handles quantities, features, parts, and assemblies, as well as systems and subassemblies, all very important notions in describing products. It is not intended to automate any part of designing *per se*, but to provide a framework for designers to think about design problems in a more structured manner and to form the logical foundations for tools to aid designers in their daily tasks. Vis-a-vis this last issue, this paper has also discussed how AIM-D is being used to develop *Designer*, a KB system intended to represent product information in a computerized form. *Designer* draws from both object-oriented and KB technologies; the operations possible in *Designer* satisfy the axioms of AIM-D, and thus preserve the logical structure of the theory.

AIM-D is not a closed theory; there remain a number of possible extensions that may be added to the language. If AIM-D is to ever be a universal theory of design information, then these extensions will have

to be addressed. The most important extensions, currently under development by the author, are the incorporation of function, and the ability to validate product models. As part of ACM (Artifact-Centered Modeling), AIM-D forms the foundation of a theoretical development that will eventually cover all aspects of the design endeavor, including design processes, and enterprise modeling.

REFERENCES

Bernays, P. (1968) *Axiomatic set theory.* Dover Publications, New York.

Bijl, A. (1987) An approach to design theory. *Proceedings, IFIP WG 5.2 Working Conference on Design Theory for CAD* (eds. H. Yoshikawa and E. A. Warman), pp. 3–31.

Brachman, R. J., McGuiness, D. L., Patel-Schneider, P. F., and Resnick, L. A. (1991) Living with CLASSIC: when and how to use a KL-ONE-like Language. *Principles of Semantic Networks: Explorations in the Representation of Knowledge* (ed. J. F. Sowa), pp. 401–56.

Buvac, S., Buvac, V., and Mason, I. A. (1995) Metamathematics of contexts. To appear in *Fundamenta Informaticae*, **23**.

Capoyleas, V., Chen, X., and Hoffman, C. M. (1996) Generic naming in generative, constraint-based design. *Computer-Aided Design*, **28**, 17–28.

Copi, I. M. (1979) *Symbolic logic, 5th ed.*. MacMillan Publishing Company, Inc., New York.

DePaoli, F., and Tisato, F. (1994) CSDL: A language for cooperative systems design. *IEEE Transactions of Software Engineering*, **20**, 606–16.

Eastman, C. M., Bond, A. H., and Chase, S. C. (1991) A formal approach for product model information. *Research in Engineering Design*, **2**, 65-80.

Fraenkel, A., A., Bar-Hillel, Y., and Levy, A. (1973) *Foundations of set theory.* North-Holland, Amsterdam.

Friedman, G. J, and Leondes, C. T. (1969) Constraint theory, part I: fundamentals. *IEEE Transactions on Systems Science and Cybernetics*, **5**, 48-56.

Fulton, J. A. (1992) Enterprise integration using the semantic unification meta-model. *Proceedings, 1st International Conference on Enterprise Integration Modeling* (ed. C. J. Petrie, Jr.), pp. 278–89.

Gonzalez, A. J., and Dankel, D. D. (1993) *The engineering of knowledge-based systems, theory and practice.* Prentice Hall, New Jersey.

Grove, A. J. (1995) Naming and identity in epistemic logic, Part II: a first-order logic for naming. *Artificial Intelligence*, **74**, 311–50.

Hakim, M. M., and Garret, J. H. (1993) A description logic approach for representing engineering design standards. *Engineering with Computers*, **9**, 108–24.

Henderson, M. R., and Taylor, L. E. (1993) A meta-model for mechanical products based upon the mechanical design process. *Research in Engineering Design*, **5**, 140–60.

Hofstadter, D. R. (1979) *Goedel, Escher, Bach: An eternal golden braid.* Vintage Books, New York.

IEEE (1991) *IEEE standard for the scheme programming language.* IEEE Std 1178-1990, Institute of Electrical and Electronic Engineers.

Jaluria, Y. and Lombardi, D. (1991) Use of expert systems in the design of thermal equipment and processes. *Research in Engineering Design*, **2**, 239–53.

Johnson, R. E., and Zweig, J. M. (1991) Delegation in C++. *Journal of Object-Oriented Programming*, **7**, 31–4.

Karnopp, D. C., Margolis, D. L., and Rosenberg, R. C. (1990) *System dynamics: a unified approach.* Wiley and Sons, New York.

Nebel, B. (1990) Reasoning and revision in hybrid representation systems. *Lecture Notes in Artificial Intelligence, No. 422.* Springer-Verlag, Berlin.

NIST (1993) *Integration definition for functional modeling (IDEF-0).* Federal Information Processing Standard Publication 183, National Institute of Standards and Technology, Gaithersburg, MD, USA.

Pabon, J., Young, R., and Keirouz, W. (1992) Integrating parametric geometry, features, and variational modeling for conceptual design. *International Journal of Systems Automation: Research and Applications*, **2**, 17–36.

Polat, F., Shekhar, S., and Guvenir, H. A. (1993) A negotiation platform for cooperating multi-agent systems. *Concurrent Engineering: Research and Applications*, **1**, 179–87.

Roozenburg, N. (1992) On the logic of innovative design. *Research in Design Thinking* (eds. N. Cross, K. Dorst, and N. Roozenburg). Delft University Press, Netherlands, pp. 127–38.

Rosen, D. W. (1993) Feature-based design: four hypotheses for future CAD systems. *Research in Engineering Design*, **5**, 125–39.

Rosen, D. W., and Peters, T. J. (1992) Topological properties that model feature-based representation conversions within concurrent engineering. *Research in Engineering Design*, **4**, 147–58.

Salustri, F. A. (1995) An artifact-centered framework for modeling engineering design. *Proceedings, 10th International Conference on Engineering Design* (ed. V. Hubka). Edition HEURISTA, Zurich, pp. 74–9.

Salustri, F. A. (1996) Integrated computer modeling of engineering design information. *Proceedings of the Canadian Society for Mechanical Engineering Forum* (ed. S. A. Meguid), pp. 613–24.

Salustri, F. A., and Venter, R. D. (1992) An axiomatic theory of engineering design information. *Engineering with Computers*, **8**, 197-211.

Salustri, F. A., and Venter, R. D. (1994) A new programming paradigm for engineering design software. *Engineering with Computers*, **10**, 95-111.

Shapiro, S. (1991) *Foundations without foundationalism: a case for second-order logic.* Clarendon Press, Oxford.

Suh, N. P. (1990) *The principles of design.* Oxford University Press, New York.

van Dalen, D., Doets, H. C., and de Swart, H. (1978) *Sets: naive, axiomatic and applied.* Pergamon Press, Oxford.

Ward, A. C. (1992) Some language-based approaches to concurrent engineering. *International Journal of Systems Automation: Research and Applications*, **2**, 335–51.

BIOGRAPHY

Filippo A. Salustri received his doctorate in mechanical engineering in 1993 from the University of Toronto. He spent one and one third years at Wayne State University in Detroit as an assistant professor of engineering. Currently, he is an assistant professor in the Department of Industrial and Manufacturing Systems Engineering at the University of Windsor, Canada. His industrial collaborations have included Henry Ford Hospital (Detroit), Ford Motors (Dearborn) and Spar Aerospace (Toronto). His primary areas of research are design theory and the development of knowledge-based systems for engineering applications.