THE FORMAL MODELING OF ENGINEERING DESIGN INFORMATION BY MEANS OF AN AXIOMATIC SYSTEM

by

Filippo Arnaldo Salustri

A thesis submitted in conformity with the requirements
for the Degree of Doctor of Philosophy,
Department of Mechanical Engineering, in the
University of Toronto

To
Mina

Abstract

There is mounting evidence in the current literature which suggests that our collective understanding of engineering design is insufficient to support the continued growth of the engineering endeavor. *Design theory* is the emergent research field that addresses this problem by seeking to improve our understanding of, and thus our ability to, design. The goal of this author's work is to demonstrate that formal techniques of logic can improve our understanding of design. Specifically, a formal system called the *Hybrid Model* (HM) is presented; this system is a set-theoretic description of engineering design information that is valid independent of (a) the processes that generate or manipulate the information and (b) the role of the human designer. Because of this, HM is universally applicable to the representation of design-specific information throughout all aspects of the engineering enterprise. The fundamental unit in HM is a *design entity*, which is defined as a unit of information relevant to a design task. The axioms of HM define the structure of design entities and the explicit means by which they may be rationally organized. HM provides (a) a basis for building taxonomies of design entities, (b) a generalized approach for making statements about design entities independent of how the entities are generated or used, and (c) a formal syntactic notation for the standardization of design entity specification. Furthermore, HM is used as the foundation of DESIGNER, an extension to the Scheme programming language, providing a prototype-based object-oriented system for the static modeling of design information. Objects in the DESIGNER language satisfy the axioms of HM while providing convenient programming mechanisms to increase usability and efficiency. Several design-specific examples demonstrate the applicability of DESIGNER, and thus of HM as well, to the accurate representation of design information.

ACKNOWLEDGMENTS

I would like to thank a number of people who have assisted me in one way or another to complete this work. First, I must thank Dr. R. D. Venter, my advisor and mentor, for his inspiration and guidance. As well, I feel deep gratitude to Drs. B. Benhabib and K. C. Smith who, as members of my Endrenyi Committee, provided valuable feedback and helped me steer clear of sand-traps; and Dr. W. R. Johnson, with whom I began my Doctorate and who introduced me to computer-aided engineering. The assistance of NSERC is also acknowledged with appreciation.

I must also thank my fiancee, Mina, whose patience with me during the last two and a half years has certainly qualified her for sainthood.

Many others – graduate students, professors, and friends – contributed too, professionally and otherwise; to all of them, my sincerest thanks.

Contents

List of Figures

List of Symbols

The notation presented in this section is drawn from accepted systems of notation in predicate calculus and axiomatic set theory. In addition, a number of symbols used exclusively by the author for the Hybrid Model (Part III) are also included. Usage of these symbols is restricted to Part III, wherein the formal statement of the Hybrid Model in the language of mathematical logic is presented.

Basic logic and set theoretic symbols are:

$\in$ Set membership; for example, $x \in I$ is read "*x is a member of I*". This is taken to be primitive to the 1st order predicate calculus and so is considered an undefined primitive in axiomatic set theory.

$=$ The identity operator, which is transitive $(x)(y)(((x = y) \cap (y = z)) \Rightarrow (x = z))$, symmetrical $(x)(y)((x = y) \Rightarrow (y = x))$ and totally reflexive $(x)(x = x)$. For example, $x = y$ iff every attribute of x is an attribute of y, and conversely. This is taken to be primitive to the 1st order predicate calculus and so is considered an undefined primitive in axiomatic set theory.

$=_{df}$ Read as "...define as...", this symbol is used to introduce definitions (e.g. $X =_{df} \{x : (x \in Y) \bullet (x \in Z)\}$, which defines X to be the set of elements occurring in both Y and Z) and is distinct from the identity operator (above).

$\subset$ Subset relationship; for example, $x \subset y$ is read "*x is a subset of y*".

$\forall$ The universal qualifier, read as "for all ...". Statements using $\forall$ are composed of three parts: the $\forall$ symbol, the specification of a variable or variables over which the

quantification is performed, and a statement which is true for the variable(s). The parts are separated by parentheses. For example, $\forall(x)(x \in I)$ is read "*for all* x, x *is a member of* I". Also, if I is the set of all integers, $\forall(x \in I)(x > 0)$ defines all positive integers, and is read "*for all* x *in* I, x *is greater than* 0".

$\exists$ The existential qualifier, read as "there exists...". Statements using $\exists$ have the same form as statements using the universal quantifier. For example, $\exists(x \in I)(x = 1)$ is read as "*there exists an* x *in* I, *such that* x *equals* 1". It is generally unclear whether this qualifier should be read as "*there exists* at least *one* x *in* $I\ldots$" or as "*there exists* exactly *one* x *in* $I\ldots$". A distinction is made in [1] between the two by using $!\exists$ to indicate the latter and $\exists$ for the former. We adopt this distinction here because the semantic difference is relevant to the development of the Hybrid Model.

$\equiv$ Logical equivalence. This rule of inference is defined by the statement $(x \equiv y) \equiv (x \Rightarrow y) \cap (y \Rightarrow x)$.

$\neg$ Logical not operator, read as "...not...", and resulting in the logical negation of its immediate consequent. For example, if x is true, then $\neg x$ is false.

$\cap$ Intersection operator, read as "...and...". The result of this operation is the set intersection of its antecedent and consequent. For example, $x \cap y$ results in the intersection of the set entities x and y. This form is used exclusively for set operations. Also, the form $\bigcap_i X_i$ means the intersection of all X_i.

$\bullet$ Boolean "and" operator. The result of this operation is true iff both its antecedent and consequent are true. For example $p \bullet q$ is true iff both statements p and q are true.

$\cup$ Inclusive union operator, read as "...or...". The result of this operation is the set union of its antecedent and consequent. For example, $x \cup y$ is the set containing all the members of both set x and set y. Also, the form $\bigcup_i X_i$ means the union of all X_i.

$+$ The boolean inclusive "or" operator. The result of this operation is true iff either or both of its antecedent and consequent are true. For example, $p + q$ is true iff (a) p is true, or (b) q is true, or (c) p and q are true. The exclusive union operator (i.e. the operator

the result of which is true if only one but not both of p or q is true) is defined by the expression $(p + q) \bullet \neg(p \bullet q)$.

$\Rightarrow$ Material implication, read as "...implies...". Classically, this is the only kind of implication used in formal logic. The antecedent of $\Rightarrow$ implies the consequent; that is, if the antecedent is true, then the consequent is also true. This operator is used when only the truth value of the antecedent is known and the value of the consequent is unknown.

$\langle \ldots \rangle$ Angle brackets denote *tuples*, which are short ordered lists treated as single units. For example, $\forall(\langle x, y \rangle)((x \in I) \bullet (y \in I))$ denotes an ordered pair of integers.

$\supset$ Superset operator. Equivalent to $\Rightarrow$.

$\supseteq$ Superset-or-equality operator.

$\emptyset$ The empty (or null) set.

$\mathcal{P}(X)$ The power set of X; i.e. the set containing all subsets of X (including the empty set).

A *function* is denoted by its name followed by arguments. The arguments are in parentheses. For example, VIEW(X, φ) is a function whose name is VIEW, and whose arguments (in this case) are X and φ. Functions may have no arguments. A function with one argument is called a *unary* function; a function with two arguments is called a *binary* function; a function with more than two arguments is called an *n-ary* function. A function returns some data entity. In general, a function is written $f(x)$.

Function variables (i.e. variables that represent functions) are written in greek characters, for example φ.

A *predicate* is like a function, but it can only return one of two values, *true* or *false*.

The following symbols are used exclusively in the Hybrid Model.

X, Y, Z individual objects.

C_i A collection i of objects.

a, b, c attributes of objects.

O The set of all objects.

T	The set of all object types.
T, U, V	types.
C	The set of all classes of objects.
A	The set of all attributes.
D	The set of all attribute domains.
R	The set of all attribute ranges.
Γ	The set of all view definitions.
γ	A view definition.
IS_A	The typing predicate, read as "...is a..." and used to relate objects to types. For example: IS_A(X, T) is true if object X is of type T, and false otherwise.
INHERITS	The inheritance predicate, read as "...inherits from..." and used to relate object types. For example: INHERITS(T, U) is true if type T inherits from (i.e. is specialized from) type U or, similarly, if type U is inherited by (i.e. is generalized from) type T.
Δ	The set of all aggregate predicates.
δ	An aggregate predicate.

Glossary

This glossary contains terms relevant to the work presented herein that are not commonly used in engineering, but that are quite common in other fields. For each term in the glossary, the page where the term is first used is given in parentheses.

Abstraction Mechanism: A device or technique whereby details are removed from some collection of information leaving only that which is considered essential. Abstraction mechanisms permit the ordering, or organization, of information. (page 19)

Cartesian Product: The cartesian product of two sets A and B is defined as the set of ordered pairs such that the first element of every ordered pair is a member of A and the second element of every ordered pair is a member of B. (page 85)

Completeness: The property of a formal system wherein exactly all true statements can be proven true and exactly all false statements can be proven false. (page 48)

Consistency: In logic, the state of a formal system containing no contradictions; that is, a formal system is consistent if all axioms and theorems in the system are *valid* (see below) with respect to each other. (page 48)

Design Entity: A unit, not necessarily realizable in and of itself, of relevance in design; an information model of real world structures of use in a design process, but not including the design process itself. (page 74)

Dynamic Data Modeling: The modeling of semantic properties and the manipulation of data structures, often in reference to database transactions. (page 157)

Encapsulation: The discretization of a quantity of information into a meaningful structure that can be treated as a single unit. (page 76)

Epistemology: The study of a theory of the nature and grounds of knowledge, especially with reference to its limits and validity[1]. (page 81)

Formal System: A system consisting of a set axioms taken as premises, and a set of rules by which theorems may be proven by application of the axioms. (page 19)

Heuristic: Something valuable for empirical research, but unproved or incapable of proof. (page 34)

Isomorphism: In logic, the relationship between a formal system and some perceived aspect of reality. A formal system is isomorphic to some real-world phenomenon if it models it correctly. (page 40)

Ontology: A branch of metaphysics relating to the nature of being; a particular theory about the nature of being or the kinds of existence[2]. (page 90)

Paradox: In logic, a statement that can be proven both true and false in a given formal system. (page 48)

Static Data Modeling: The description of data objects and their relationships without considering of the operations in which such structures may be used. (page 111)

Universe of Discourse: The domain about which all interesting arguments are made. For example, in set theory, the universe of discourse is that of all sets. (page 74)

Validity: In logic, the state of being true under any interpretation; that is, a statement or formula is valid if it can never be false. (page 38)

---

[1] From Webster's 7th Dictionary.
[2] From Webster's 7th Dictionary.

Part I

INTRODUCTION

Chapter 1

Introduction

1.1 Preamble

What is engineering design?

In the broadest sense, the issue addressed by this work is that humanity lacks a sufficient understanding of the process of engineering design. Although simple enough to phrase, this problem has been an issue of contention for years, and the sometimes very emotional arguments made by various proponents of one viewpoint or another, methodology A or B, system X or Y, have possibly done more harm than good. It has always struck this author as highly suspicious behavior when one is unable to refrain from overt generalization and emotional rhetoric to convince an audience. Unfortunately, members of *all* the various schools of thought involved in the debate may be accused of this kind of behavior, leaving one with the distinct impression that no one is completely right.

However, the unavoidable facts are these: engineering design has existed in one form or another since ancient times. In this interval, design has changed – *evolved* – not only in response to our ever-increasing understanding of the physical universe, but also in other, relatively arbitrary ways, responding to forces not particularly natural: sociological, psychological, environmental, governmental, and political. All these forces have had a hand in shaping design as it is now, and their continued influence has required designers and design researchers to adapt to their exigencies.

The one constant throughout the history of engineering design has been that the requirements placed on designers and on the products they design have continually increased in complexity. Similarly, as we gain more knowledge about the universe, that much more information can be applied to the problems we have to solve. The complexity of most current-day design problems is many orders of magnitude greater than that of problems faced by the designers of the last century, and there is no reason to suspect that this trend will change in the future. Correspondingly, in order to meet the challenge of today's design problems, we must learn to manage all the information at our disposal in an efficient, concise and timely manner.

Design researchers are thus faced with the task of dealing with this complexity; simply ignoring it is unacceptable. In response to this challenge, a new field of research has emerged: *design theory*. Its goals are (a) to respond to the increasing complexity of engineering design problems, (b) to increase confidence in our solutions to the design problems of the future, and (c) to overcome limitations and difficulties associated with the process of design by providing more consistent, logically structured methodologies and techniques.

Furthering our understanding of design is a worthwhile goal because it leads to verifiable explanations of the phenomenon of design. Historically, explanations of this kind in other fields (such as physics, medicine, etc.) have led to progress, advancement, and new insights into the nature of the phenomenon.

Many segments of the design process are still based in large measure on arbitrary, *ad-hoc* decisions and processes. Still, design theory has provided a focal point for the efforts of hundreds of researchers, and their work has already made notable contributions towards the goals stated above. It marks the beginning of a new stage in the evolution of design, a renaissance of sorts; we, as designers, are collectively and for the first time attempting to examine our role in society, and to examine design itself critically, objectively, logically. Holding design theory as a central concept, researchers are striving toward a level of formal rigor in engineering design, a certain scientific legitimacy that has, to date, been rather elusive.

In the recent past, engineering design has been considered largely an "art" or even a "skill", an endeavor not amenable to formalization and scientific scrutiny. This is changing: the introduction of expert and automated systems, quantitative cognitive design research, new methodologies such as concurrent engineering, new forms of mathematics (for example, fuzzy logic) and other technological and scientific innovations are permitting a new view of design to develop, a view in which technology and creativity, science and intuition are linked in a symbiotic relationship, forming a whole that is greater than the sum of

its parts. It is not unlike the Renaissance, during which time science grew from a highly mystical practice to the much more rational, reasonable, and far more useful endeavor, it is today. As this maturation process continues, we will have to examine the engineering design enterprise and possibly change our ideas about what it really is. In order to be successful, every effort must be made to maintain a flexible yet definitive framework within which design and design theory can evolve in a formal manner.

It is this author's humble desire to contribute to this endeavor, and to help the engineering profession develop to its next plateau.

1.2 Premises

The relevance of the work presented herein depends on the acceptance of certain premises. These premises cannot be proved in any mathematical or logical sense of the word because they are *extralogical*, that is, they are empirical, deriving directly from observation of the state of design and of reality. As such, the most that can possibly be expected is to provide sufficient empirical evidence to suggest an acceptable degree of confidence.

These premises are discussed here, at the outset, so as to define the boundaries within which the rest of this work exists.

Design is not well-understood. There is no clear, precise definition of what design is. It has, for example, been referred to in the recent literature as "...the evaluation and satisfaction of many constraints..." [2]; "...planning for manufacture..." [3]; a largely intellectual, cognitive process [4]; an "...ill-defined art...which lacks formal definition..." [5]; a somewhat "mystical" process [6]; a "...socially mediated process..." [7]; and many others. All these descriptions are, to be sure, partly right. But the totality of what is involved in design is lost in each case. Certainly, many researchers have been motivated to perform design research expressly due to the apparent lack of current understanding [8,9].

Design is not currently efficiently performed or taught. In a recent report [10], the National Research Council (USA)[1] has taken the position that engineering design education is weak, and that this weakness

[1] in collaboration with the National Science Foundation and other bodies

is preventing the best available design practices from reaching industry. This point of view has been advocated widely, both in the United States, and in Canada [11, 12]. Educators must find new ways to communicate effectively to students if these students are to someday design effectively. But without a good understanding of what design is and how it is performed, such necessary communication is impossible. Thus, the burden finally lies with researchers, who must advance our basic understanding of design.

Formal theories of design would lead to improved practical methodologies. A formal theory is a logical, objective system for the maintenance of knowledge and the investigation of phenomena. Formal theories are the cornerstone of both the natural and engineering sciences. The establishment of formal theories of design may provide the benefits to designers that they have provided to scientists. This notion has been echoed by other researchers (e.g. [13, 14]).

It is possible to formalize at least segments of the design process. A number of methodologies and techniques already exist, defined in at least informal terms, that have provided considerable insights, if not into the nature of design itself, then at least into the more practical aspects of the engineering endeavor. The structure currently used every day in design processes all over the world did not exist a century, or even a decade, ago. There is no indication that we have yet exhausted all the directions in which such a structure can extend. Whether the design process as a whole can be entirely automated remains an open question; still, there are many areas where increased formalization is possible and desirable. The advantages are numerous: improved communications and tools for teaching, more reliable analysis of designs for correctness, information integrity and shorter development times are but a few [15].

1.3 Statement of Thesis

The thesis of this work is: *axiomatic set theory provides a basis whereby design information can be rigorously specified independent of design processes giving rise to or otherwise manipulating that information.*

This statement captures the essence of a number of arguments, all of which will be presented and examined in this document. Such terms as *axiomatic set theory*, *design information*, *design process* and

so on will be defined where possible in accordance with commonly accepted practice; in some cases, where conventional definitions are vague or imprecise, an attempt will be made to specify them more fully. Some terms, such as *isomorphism*, have specific denotations in logic; since this work focuses on the use of logic, we will use the logical rather than engineering definitions. A glossary of important terms is provided at the beginning of this document.

The heart of the work focuses on the derivation of a formal system called the Hybrid Model of design information (HM); the system consists of nine axioms and various definitions and theorems. The axioms capture essential properties of design information within a logical framework, including five *abstraction mechanisms* that permit the organization of information according to various criteria. The axioms will be presented in the notation of mathematical logic; explanation and discussion will be presented in English.

In order to demonstrate the application of formal systems to practical issues in engineering design, a new programming language for engineering computation, called DESIGNER, shall be presented. DESIGNER relies on HM for rigor. It represents a computational model that captures arbitrary information about specific design entities and permits the organization of that information; it is *not* intended to capture methodological or procedural information about the design process itself. It will be shown that a strong continuity of logical rigor exists from HM through to the implementation of DESIGNER. Various examples specific to engineering are included to demonstrate DESIGNER's capabilities.

Given the novelty of design theory as a field of research, a significant body of fundamental work does not yet exist. This is indicated in the premises (Section 1.2). The author's work must remain somewhat general, if for no other reason than this. In the process of developing the theories and ideas presented herein, the author has also developed a number of collateral notions which, though not directly associated with the precise topic of this work, impact upon it in an ancillary capacity. This material is nonetheless relevant and original, and is included in this document for the sake of completeness.

With this in mind, the author also defines the thesis of this work in a broader sense; namely that *the use of* **formal logic** *can significantly improve our understanding of design, and provide a framework within which highly effective tools for managing the complexity of design can be generated.* The answer to the question of the specific thesis of this work, stated at the beginning of this Section, will be used to corroborate this more general statement.

It is noted that this dissertation directly addresses what is given in [4] as the first mistake of current

design education: "We have done very little research to develop a fundamental scientific understanding of engineering design processes."

1.4 General Remarks

As design theory continues to gain acceptance and popularity, the organizational aspects of these research efforts will become increasingly important. Successful organizational techniques rely on the detection of, and accommodation to, the structure that exists within information available to designers. The creation of these techniques must occur in a *logical* fashion. Paraphrasing Quine [16], the truths of logic may be reckoned among the truths of design, thus making logic an essential basis of its formal understanding. Throughout this work, arguments will be structured in as logical a manner as is possible, and logic will be invoked often as the basis upon which we will make our observations.

Efforts in design theory have rightly depended on the use of computers for their capacity to store information accurately and to calculate and maintain the complex relationships that exist between data. As is noted in [17], "...the human brain is better able to *recognize* than recall...." Computers, with their relatively infallible memories, are of great assistance in this regard. Furthermore, as the world economy continues to move from a product and service base to an information base, information management issues in design will become more important [7, 18–20].

Design research in the recent past has examined the use of databases, application-specific languages and expert systems, as well as more theoretical studies in such areas as constraint satisfaction and symbolic computation. These tools have been applied with varying degrees of success to component assembly [21], design exploration [22, 23], solid modeling [24], finite element analysis [25], etc. The unique nature of design suggests that generalized information management approaches will not necessarily support all of its aspects [26–28].

Currently, however, there is growing concern regarding the *semantics* of engineering design. Many recent research efforts have met with limited success because not enough is understood about the *meaning* of the information we use. The understanding we do have tends to be empirical and intuitive [29, 30] and its organization is neither particularly structured nor logical. In response to this, researchers have begun to backtrack, seeking a return to sound, logical first principles in design. Two notable examples of this trend

are [4, 31]. The notion of the existence of formal first principles for design has guided the author's work presented herein as well.

The importance of the *organization* of design information cannot be overemphasized. This issue is strongly tied to the search for semantic formalization. To organize information means to order it. The imposition of order on information is identical to the extraction of meaning from it, and makes *explicit* such information as would otherwise be *implicit* only. Increasing the amount of explicit information present in a collection of data decreases the amount of *interpretation* that must be performed to extract its semantics [32, 33]. Therefore, the study of organizational schemes for design information is synonymous with the study of its semantics. The theme of organization pervades the author's work.

The search for a formalization for engineering design must necessarily be conducted in a logical, scientific and internally consistent manner [4]. For this purpose, the author has selected axiomatic set theory and discusses it briefly in Section 8.2. The role of logic in design is critical to the development of the author's theory, and will be discussed in detail before the actual theory is presented.

This document is arranged in five Parts. The central three Parts form the body of the work, beginning with the most general and theoretical remarks, and proceeding towards more specific matters. Part II will focus on the role logic plays in design, indicating some of the short-comings of the current understanding of design and indicating how logic can help resolve these problems.

Part III deals with the central thesis and presents the Hybrid Model (HM) of design information. Naturally, no discussion about design information can be carried out without *some* reference to the design process, but the author will demonstrate that it is both reasonable and advantageous to separate information about a design artifact from the actions carried out on or with this information. It is essential to understand what forms of information are available before any meaningful discussion regarding design *processes* can occur. Thus, *the theory deals specifically and only with design information.* The design process will be discussed only insofar as to define the design information management problem. Issues such as concurrent design are not addressed because they are aspects specific to the design process; that is, they affect how information is manipulated, but not the information itself.

Part IV will describe a new programming paradigm devised by the author, built upon HM. The intention is to indicate the immediate benefits that can be reaped from a formal theory such as HM, and to provide a testbed with which further research in design theory may be conducted. Various design-specific examples

will demonstrate the capabilities of the resulting computational system.

Finally, Part V will conclude the work with a general discussion of results and directions for future research.

A glossary of important terms and a list of symbols are provided at the beginning of the document.

Chapter 2

Literature Survey

2.1 Introduction

This Chapter will present a survey of recent literature examined in the course of performing the work reported herein. Due to the relative youth of design theory as a research field, literature directly within the field is not abundant. However, the author has found a great deal of information in the peripheral areas of formal logic, computer-aided engineering, concurrent engineering, and computer science which is relevant to the matter at hand.

Because this dissertation is squarely within the field of design theory, comments that will appear in this Chapter shall be biased in that direction. In other words, the primary criterion for the evaluation of other research shall be the degree to which it contributes to the overall understanding of design. In some cases, this contribution will be slight; this does not mean that the surveyed work is of no value, but only that it is of limited use strictly within design theory itself. Given the nature of this survey, literature dealing with theoretical aspects of computer programming, as well as fields such as formal logic and others will not be dealt with directly.

The author has found that the existing literature can be divided coarsely into three categories: design theory, computer-aided engineering, and cognitive and concurrent design research. These three categories are not decoupled; some cross-over is bound to occur as techniques are applied to various domains

(for example, the application of a specific design theory to the generation of a computerized designer's aide). However, principal contributions by various researchers can be categorized on the whole using this scheme; it shall be used here to help organize this survey.

2.2 Design Theory

Surveyed work which the author classifies as design theoretic in nature is hallmarked by three properties. First, computer technologies are not essential to the contribution of the work; that is, the principal contribution is not simply a new computerized designer's aide, database, or other program. Second, design theoretic work aims at a unified view of design in the most global sense, not only as it affects some very specific task within design (for example, synthesis and analysis of mechanisms [34]). Although such work may contain strong theoretic elements, it is not design theory because it deals with specific tasks and excludes issues of integration with other aspects of the design endeavor. Third, the role of the designer is not central to the development of the work; this excludes issues of cognition, intuition, judgment, etc.

The efforts of two researchers in particular are exemplary of the work currently being done in design theory:

Nam P. Suh has recently published a text [31] wherein he sets forth an axiomatic theory of design. Suh assumes that a design problem can be stated in terms of *functional requirements* (FRs) and that its solution, a design artifact, is defined in terms of a number of *design parameters* (DPs). He argues for this approach based on the empirical evidence of how design problems are commonly stated, and how their solutions are commonly specified.

Based on this assumption, his theory contains only two axioms, eight corollaries and sixteen theorems. We will examine only the axioms here, since the corollaries and theorems may be derived from them. The first axiom is that the FRs of a design problem should be independent of each other (i.e. uncoupled). He argues that coupled functional requirements indicate some misconception of the design problem; several examples are provided to support this notion. The second axiom of Suh's theory is that the information content of a design specification should be minimized. The intention here is to ensure that there is no duplicated information or information arising from coupling between FRs. It is shown that, all else being equal, a set of uncoupled FRs leads to a "simpler" design (i.e. having a minimum required information

content) than an equivalent set of coupled FRs.

Suh's theory does not formally define an exact procedure to be followed in an arbitrary case in order to find an appropriate solution, but rather defines a *collection of rules* that a designer can use to analyze a design problem and guide evaluation and subsequent iterative redesign of a solution until it becomes satisfactory.

Although the theory is for the most part presented in English, a chapter is reserved near the end of the book for a restatement of the theory in the language of predicate logic. This author notes that though the statement of Suh's theory is amenable to representation in predicate logic, a number of important concepts are defined as primitive predicates only informally (such as *feasibility*, *measure of information content*, and *coupling*). Since logic is a field wherein an argument is only as strong as its weakest link, such informality at the very outset of the presentation can only be taken as a shortcoming of the theory.

Nonetheless, the text is replete with examples ranging from the design of a can opener to the re-organization of the Engineering Directorate of the National Science Foundation, and is an ideal vehicle for the introduction of logical structured thinking processes into the field of design, both as an educational tool and as a general reference.

The second principal contributor to design theory is John Dixon (*et al*) who, in numerous papers and articles (e.g. [9, 13, 29, 35, 36]) strongly advocates a more "scientific" approach to design and design research. Rather than the generally philosophic approach taken by Suh, Dixon's work centers on experimentation with computers to prove or disprove general notions relevant to design. In [9], for example, a taxonomy for various kinds of design problems is described. The criterion for ascribing a particular problem to one kind or another is based on the nature of the initial state of knowledge when the problem is defined, and the final state of knowledge at the problem's solution. For example, *phenomenological design* is characterized by a function to be supplied and a physical phenomenon that will provide that function. Each kind of design problem thus recognized, argues Dixon, suggests a class of solution methods. Thus the taxonomy of design problems is seen as leading to a corresponding taxonomy of design methodologies. As these taxonomies become more detailed, Dixon also reports on various software systems devised to satisfy taxonomic and other requirements. The systems are then used to determine what advantages are provided, if any, by the approach. The main contribution of Dixon's work is seen by this author as the explanation of the nature of various kinds of design problems and the classification of known and newly devised solution

techniques based on experimentation using computers. Many of Dixon's classifications are quite coarse, but since no other taxonomic systems have yet been widely accepted, he has little alternative. Two other classifications that he and his co-workers have advocated include: the separation of design into (a) design problems, (b) the people who perform design and (c) the environment within which design occurs [37]; and categorization of design theories as *prescriptive*[1] (tending to describe *how* design should be carried out), cognitive-descriptive (describing *what* design seeks to achieve) and computational (formalization through computerization) [38]. These systems all seek to provide flexible tools to guide researchers towards a better understanding of the nature of design, rather than a series of rigid, inflexible structures defining exactly the nature of design as a process.

Insofar as purely theoretic research is concerned, special comment should be also made of three other efforts:

The first is the Extended General Design Theory proposed by Yoshikawa in [39]. The importance of this work is three-fold: firstly, it was a fairly early attempt to employ some techniques of logic to discuss the nature of design; secondly, it sought to place design in a more global framework and discussed the relationship between design and such fields as physics, philosophy, and technology, by representing the universe as perceived by man as divided into three distinct domains: *logical*, *conceptual* and *physical*; thirdly, Yoshikawa was one of the first to advocate the distinction between *what* design is versus *how* design is performed. From this starting point, Yoshikawa began investigating the nature of design, thus including from the very outset the notion that design must occur within an environment, by which it is affected and which it affects. This view of engineering as part of the human experience has more recently become quite relevant in nationally mandated efforts to improve design as an endeavor [10].

The second is the notion of *recursive design* as proposed by Ward [40]. Design is seen as recursive (rather than *iterative*) in cases where the design problem may be broken down into smaller and smaller components by the recursive application of some single methodology. This differs from the more conventional iterative approach in that an iterative design process is applied many times to a detailed design alternative, its goal being convergence of successive solutions when compared to an externally defined set of criteria. The advantage of recursive design is its ability to implicitly handle a wide number of design alternatives, something that typical iterative techniques do not do well. However, if the resulting tree structure of all

[1] sometimes referred to as *normative*.

alternatives in a recursive design process is not carefully "pruned" to eliminate unacceptable designs as early as possible, it can lead to an intractable number of solutions. The notion of recursive design is quite new, but shows some potential, especially in concurrent engineering environments.

The third and last notable theoretic effort is that of Fauvel [41]. In his approach to modeling design, no emphasis is placed on any one aspect. Rather, two abstract notions, *embodiment* and *activity*, are used to model any design entity. An activity is any procedure or methodology used by a designer; an embodiment is the resulting effect of that procedure or methodology on a design. This essentially functional approach is particularly flexible and capable of providing an integrating framework for the overall design endeavor. It is also reminiscent of the model developed in this thesis (see Section 8.1).

Dixon is also a leading proponent of so-called *feature-based design* (e.g. [42–45]). A feature is generally considered to be a modeling entity more immediately relevant to design and manufacturing than a simple solid or other geometric model. It is often used in connection with efforts to integrate design and manufacturing. However, even Dixon himself states that the exact nature of a feature has yet to be defined [42], and other researchers (e.g. [46, 47]) have demonstrated that feature-based approaches are not computationally feasible over large feature sets. Nonetheless, the importance of features as modeling structures makes it worthy of note here. Unfortunately, researchers have almost invariably treated features as the basis for the construction of software systems rather than legitimate conceptual structures to aid in the generation of design theories; nonetheless, since we are interested in design theory and not engineering computation, we discuss features in this survey only insofar as they represent a conceptual tool useful in design theory. Dixon is one of the few who have used the notion of a feature to guide the development of his design research. Features have found use in the study and generation of taxonomies of design entities [48, 49], formal languages for the specification of spatial relationships [50, 51], and the integration of design and manufacturing [52–54].

Another area wherein a great deal of work of a design theoretic nature has been done is the area of *constraints*. A constraint is generally defined as some kind of relationship between variables or parameters that restricts the set of acceptable values that the variables may have assigned to them. Constraints capture a restriction placed on a design by the nature of the design problem; thus all constraints on a particular design must be satisfied for the design to represent a possible solution. The constraint satisfaction problem has been determined in the general case to be *NP-complete* [55, 56]; that is, all but trivial cases

are computationally intractable [57]. Constraints also affect design optimization and simulation [58–62]. Thus, finding alternative strategies for dealing with constraints remains an active research area.

The first real attempt to treat the notion of "constraint" as a modeling tool mathematically appears to have been by Friedman and Leondes [63–65], wherein not only a general mathematical treatment of *constraint theory* was given, but various kinds of constraints relevant to the engineering domain were examined. Since then, work in the theory of constraints has continued in two principal directions: computation and theory. The computational aspects of constraints will be discussed in the next Section. Insofar as the theoretic aspects of constraints as concerned, many efforts have been directed in the use of constraint theory to *optimize* design; notably, the work of Wilde [66] who introduced the notion of *monotonicity* and established several principles by which monotonicity can be used to optimize mathematical models of design artifacts. All this work has been continued by many others, including [2, 22, 58, 67–72]. Also, various specific aspects of constraint theory as applied to design have attracted the attention of researchers: the role of constraints in discrete event systems for modeling design processes [73, 74]; the representation of spatial constraints to model shape and physical structure [75, 76]; categorization of constraints types [77]; and the use of constraint networks as models of design processes [78].

There are many other research efforts that have contributed to the development of design theory; to even mention them all is an intractable proposition. However, a glimpse of a few can indicate the depth and richness of ongoing research of potential relevance:

Various mathematical and logical forms have found their way into design theory, with the goal of advancing the integration of otherwise disjoint aspects of design, including: probability and fuzzy logic applied to the representation of uncertainty in the design process, especially in *top-down* approaches [47, 79]; predicate calculus [80] applied to the capture of design knowledge [14]; and integration through the use of information management techniques and information theory [81–83].

The development of general methodologic frameworks for particular sub-domains of the general design process has attracted considerable attention, including Design for Manufacture (and Assembly) [84], design for quality [85], generalized techniques to assist in the organization of design knowledge for the sake of simplifying its complexity [86–88], and efforts to create viable taxonomies or classification systems for design problems, methods or entities [17, 89, 90].

2.3 Computer-Aided Engineering

Into this category falls the majority of the work (over 50%) of which the author is aware, and consists of models and actual implementations of computerized design aides. Efforts in computer-aided engineering often have implicitly defined within them models or portions of models of design in general. This has been necessary because formal bases for design do not yet exist. Indeed, the need for formalisms for design information and processes is often regarded as an essential prerequisite for achieving *integrated* computerized design systems [91,92]. Such conceptual models and formalisms guide the development of software systems. The motivation for the development of the conceptual model arises primarily from the requirements of software design, rather than from the requirements of design in general. In many cases, this has resulted in confusion between the modeling requirements of design and the exigencies of computer programming. We defer a detailed examination of this issue to Section 4.3; here, we shall examine the breadth and nature of recent work by other researchers. As indicated in the introduction to this Chapter, emphasis shall be placed on contributions to design theory rather than to computer-aided engineering.

The most noteworthy effort is that of Charles Eastman *et al* [93–98]. This work presents a new conceptual data model, called the Engineering Data Model (EDM), that defines classes of designed products called *product models*. The content of a product model corresponds to a design database schema (i.e. the organizational structure of a design database, not necessarily the actual data within it). One of the major advantages of EDM is that it is entirely independent of both hardware and software considerations; this separation greatly simplifies the system, permitting clearer definition of important notions for design without the need for actual computation. Additionally, notions of formal logic are used as the base upon which EDM is built; this provides rigor to permit the "correctness" of a particular product model to be investigated. EDM product models are defined entirely in terms of three primitive constructs (domains, aggregations and constraints) plus several higher level constructs built up from the primitives.

The important contribution of Eastman's work from the point of view of design theory is that EDM provides a formal structure for design information, albeit for the express reason of generating database schema. This structure could be used to examine the nature of design information itself. In this regard, EDM is unique in all the work of which this author is aware for its completeness and rigor.

However, there are two shortcomings in EDM that this author feels are significant. Firstly, EDM is a *descriptive* (or declarative) form, as opposed to an *axiomatic* form. EDM is meant to permit, among other things, reasoning about product models, but a descriptive form does not provide the apparatus to perform this reasoning in a formal manner [99]. As well, descriptive forms have been found to be domain-dependent, which can lead to difficulties in computability [100]. Axiomatic systems, on the other hand, inherently provide all the apparatus needed to check for correctness (i.e. the notion of "proof" of a model) and systematic model construction.

Secondly, although EDM remains quite detached from the exigencies of computer programming, the fact that it is a *data model* requires it to treat such issues as identity (i.e. naming, equality, etc.) from a computational point of view. This introduction of computational issues can unnecessarily complicate investigations geared to the study of design information in general.

Another notable contribution is that of Crawford and Anderson [78,101], in which a computerized system for general modeling of design is presented. The key advantage is that the system is capable of modeling solution processes as well as design problems themselves. In this way, a higher level of unification is achieved than in other efforts, and different categories of solution techniques can be established to assist in solving novel design problems. However, as is typical in these efforts, the strictly descriptive attitude taken in the work limits the use of the proposed system to the study and analysis of designs. Also, the connection between the proposed system and formal bases (e.g. logic) are not examined, thus raising questions as to whether the validity of the system can be demonstrated.

The need to express structure as an essential property of data in a computational environment has encouraged the development of various taxonomies, including taxonomies for design decisions [100], for mechanical systems [17], for design tasks [102], and for semantic operations on design knowledge [29, 103]. As well, to capture the procedural aspects of design, numerous models of design have been suggested, including meta-model evolution [102], the molecular data model [104], the state/transition model [105], the structural data model [106], and multi-layered logic [107]. These are noteworthy because they all represent *modifications* or extensions of existing data models (e.g. predicate calculus [80], the relational model [108], the entity-relationship model [109, 110], semantic data models [28, 111], etc.) leading to the conclusion, supported by many, that conventional data modeling techniques are insufficient for design. This underscores the need for additional design theoretic research, since we cannot depend on

existing schemes to provide the needed structure.

One data modeling technique deserves special mention because it was originally designed to provide support especially for non-conventional applications such as engineering, and because it has become the most popular approach to managing information: *object-orientation*. Though a detailed discussion of the nature of object-orientation is unwarranted here, it is quite relevant to distinguish it as possibly the most promising approach to design information management yet devised. In particular, the object-oriented approach specifically addresses the short-comings of its predecessor, the relational data model [108]. The relational model was conceived to address the needs of business and commercial applications, but as many design researchers have indicated, design's data modeling requirements are quite different from those of other application domains [25,93,112–118]. In particular, object-orientation is seen as providing a far richer set of abstractions for the construction and organization of design models, independence from implementation issues, reflection (for automatic analysis of design models), unified language definition for both programming and database applications, and so on. From a design theoretic point of view, object-orientation can permit the computer to become a more useful tool for investigative studies into the nature of design by hiding many of the more mundane and irrelevant issues of computation from the user.

It must be noted here that of all aspects of design, conceptual design, arguably the most important aspect of the design process, is the least well understood of all. This is made abundantly clear by the failure of all attempts to computerize it [17,60,119,120]. One tool has been indicated, however, as a possible solution to this particular problem: parametric design. In parametric design, details of various components, assemblies, etc. are ignored for the sake of capturing in parameterized form the essential attributes of design entities. In this sense, this author suggests the term *schematic design* may be more appropriate as it carries a more direct connotation of an abstract nature and of the intention to capture only those aspects that are essentially representative of the entities being designed. Parametric design has been investigated in detail by others, including [61,75,121,122]. We note, however, that two issues regarding parametric design remain problematic, especially from the point of view of design theory. First, parametric design does not permit cyclic relationships to exist between data; yet the existence of such structures has been indicated in constraint networks, especially in conceptual design [122]. Second, though parametric design may solve the problem of conceptual design, it has been demonstrated to be too restrictive for use in optimization [61]. Whether these problems indicate a shortcoming in parametric design, or a deeper

inadequacy in our understanding of conceptual design, has yet to be determined. One possible alternative is *variational design*, which adopts a different mathematical formulation than does parametric design, and which has been shown to manage cyclic constraint networks well.

Many other computational approaches have also led to insights into the nature of design. Efforts making use of systems engineering [83, 101] have demonstrated the usefulness of modular approaches to control complexity. Various research projects involving the construction of database systems for design have introduced the notion of *abstraction mechanisms* as techniques to organize and integrate our understanding of design in general; these include structural entities like features [2, 123], task-specific views of information [56, 112], and hierarchies, aggregations and other classification forms [106, 124–126]. Research into the practical aspects of constraint management have provided numerous approximations for the solution of the constraint satisfaction problem (see previous Section) [23, 24, 105, 127–130].

2.4 Cognitive and Concurrent Design Research

The third category of research is distinguished by a concern for understanding, in whole or in part, the role of the human in design; that is, rather than being concerned directly with design itself, workers in this area are concerned with the mental functions of human designers when they perform design tasks. Similarly, some researchers (e.g. [39, 120]) have distinguished between *how* design is performed (by humans) as opposed to *what* design is. Although not commonly considered together, cognitive design research and concurrent design do share an interest in the actions of the designer.

Cognitive design research is concerned with what might be referred to as the *psychology* of design, and seeks to explain, or at least quantify in some manner, the particular mental processes a designer may use. The author also includes expert (and other knowledge-based) systems research in this category, since these systems model the designer's ability rather than design itself or some aspect of it.

Concurrent engineering, on the other hand, is concerned with *sociological* issues. The principal tenet of concurrent engineering is that the involvement of all interested parties in a design process from the outset can markedly improve the decision-making abilities of the group as a whole. This kind of design is far more information- and coordination-intensive than conventional, purely sequential design processes and thus requires a much more refined strategy to assure efficient, accurate and timely communication

between team members. This requirement has led to great activity in the use of computer tools to help document design processes and communicate information between designers. Thus, again, the role of the human in the design process is the central concern.

2.4.1 Cognitive Design Research

This author has several reservations as to the relevance of cognitive design research to design theory. This issue will be dealt with in depth in Part II. For the current discussion, an abbreviated version is sufficient. Very little is understood of how the human mind functions; additionally, the outward signs of mental function (speech, gestures, etc.) do not necessarily relate directly to underlying cognitive process. Indeed, there is some evidence to suggest that "thinking" as we normally consider it is a purely *unconscious* process that we may never directly observe or even experience [131]. This detachment suggests that too much may interfere with any attempts to control the cognitive function so that it may be observed in a scientific manner. One is therefore left doubtful of efforts involving the use of artificial intelligence techniques to create expert systems and other knowledge-based systems for design which are all rooted in the presumption of some basic understanding of the human cognitive function.

However, it must also be said that there have been notable contributions to design theory made not necessarily from individual efforts in cognitive design research, but rather by the whole of the endeavor. Specifically, the various efforts of which the author is aware in the recent literature [8, 21, 24, 102, 103, 132–136] are supportive of the following observations:

First, expert systems suffer from a phenomenon called *combinatorial explosion* when applied to very wide application domains. This means that the amount of information that must be managed by these systems becomes intractably large as more and more different kinds of problems are included. However, for very specific domains, expert systems have been known to generate reasonably efficient solutions. This suggests, as has been noted in [102, 103], that though the original goal of expert systems as the ultimate design tools may never be achieved, they may be very useful as smaller components of large, integrated design systems (e.g. the design of cams [21]). As well, a very significant dependence on the structures used to represent information is indicated.

Second, expert systems, like human designers, require a certain period during which they are "trained-up"

for the kinds of tasks they are to perform. During such training periods, the system will not perform correctly since it is still learning. The resource drain that must necessarily occur during training of expert systems has not been addressed in any of the efforts of which this author is aware. As well, the information base upon which expert systems draw during this period is often *heuristic*. Heuristic knowledge is empirical in nature, often unprovable and usually based on the knowledge of particular experts in a field. So, while expert systems may perform more rapidly than humans, and are less likely to be the source of simple mechanical errors, they will carry with them all the inaccuracies of a human designer.

Third, the notion of *knowledge* is not precisely defined. The term has been associated with both the act of learning and its results[2]. This author is surprised that such a vaguely defined notion has been used to motivate the creation of so many computer software systems (i.e. *knowledge-based* systems). A fundamental aspect of these systems is that they can operate in various "intelligent" ways on information; one might suppose that knowledge is then tied to the process of *using* information. But again, we lack understanding of how the mind acts on information; we must therefore suspect any effort to mimic this behavior computationally as being based on rationalizations rather than real scientific understanding.

Fourth, as indicated in [24], most design "knowledge" imparted to expert and other such systems tends to be *routine* knowledge. This would seem to indicate, then, that expert systems would be unable to cope with new kinds of design problems. This has been indicated not only in the existing design research, but also in a more general sense in artificial intelligence [137–139].

2.4.2 Concurrent Engineering

Concurrent engineering has enjoyed significantly more success than has cognitive design research. As stated above, the goal of concurrent engineering is to parallelize the design process, bringing upstream various functions normally left until late in the design process (e.g. assembly planning). Engineering establishments that have adopted concurrent techniques have boasted marked savings in time-to-market, development and production costs, and wastage [100, 117, 140, 141], in some cases exceeding 50%. The degree of savings has surprised many, and caused a number of researchers to investigate concurrent

[2] Based on Webster's 7th Dictionary.

engineering with the aim of identifying exactly how these savings are achieved. This work is on-going and very few results of consequence have been reported. However, researchers have identified that *communication* of information is likely the key to effective concurrent engineering.

Effective communication in a technical field such as design requires a formalized notation for the specification of information (i.e. a *language*) and a flexible yet strict framework within which information can be arranged and organized [107]. This has led to the adoption of *hypertext* (also called *multi-media*) as a key computational tool for concurrent engineering research [32, 135, 142–144]. In essence, hypertext is a modeling technique for information upon which no organizational schema is available *a-priori*. Thus users of hypertext systems not only supply the information but also the various "hooks" by which the system can organize the information. The important contribution of such systems is that information that would otherwise have been only implicit (because no schema exists to capture it) can be made explicit [81]. Here is one possible use of expert systems to aid designers, though indirectly. A suitably trained system could examine a hypertext database and organize it according to various other schema identified as significant in a design environment. This idea has yet to be researched.

Finally, because concurrent engineering places great importance on the conceptual stages of a design process, a number of cognitive researchers have investigated the nature of communication between group members in concurrent engineering teams [7, 47, 135, 145, 146], and have identified various techniques of "negotiation" and sharing of knowledge whereby group decision-making can be assisted by various formal techniques (e.g. case-based reasoning [145]). It is noted in closing that cognitive research has had a more noticeable impact on concurrent engineering, possibly because of the increased availability of externalized evidence (communication between group members) of the design process.

2.5 Summary

This Chapter has presented a survey of recent literature on design theory and the associated fields of computer-aided engineering, concurrent engineering and cognitive design research. On the whole, the body of work, though not particularly voluminous due to the relative youth of the field, clearly indicates a preoccupation with the integration of the various aspects of design. Some research has dealt with specific theoretical aspects of design (such as constraint satisfaction), whereas other efforts have been larger and

less detailed in scope, seeking a general framework where particular efforts may be seamlessly combined. Some successes have been demonstrated as the results of this research percolate from research institutions into industry, but there is as yet no consensus as to the final form of a fully integrated design endeavor. Nonetheless, the successes achieved to date indicate that the principle of an integrated view of design is worthy of further study.

Part II

THE ROLE OF LOGIC IN DESIGN

Chapter 3

Introduction

The approach used in this work is quite novel: though tools of logic – such as set theory and predicate calculus – have often been invoked in supporting roles in design theory, there is no evidence of any work in design that takes logic as its sole foundation. The author finds that the uniqueness of this approach warrants some explanation, and that the explanation itself can go a long way to clarify the nature of engineering design, and indicate ways in which more robust formalisms can be achieved.

The purpose of this Part is twofold. First, the general role of logic in design theory will be discussed. Some of the problems confronting design theorists will be examined, and possible solutions based on the application of logic will be considered. A fairly wide range of topics will be covered, but the underlying notions are few and distinct. Second, a number of terms and notions of logic that are not particularly well-known in design – but that will be used extensively in the Parts to follow – will be introduced and discussed in detail.

Some important notions should be introduced before any other consideration, because of their importance in the sequel. First, we define *logic* as "...the study of methods and principles used in distinguishing correct (good) from incorrect (bad) arguments [80]." Indeed, a definition of logic that is both precise and compact is difficult to find since such a definition would to some degree depend on logic itself for correctness, and, as shall be shown, the validity of such self-dependent definitions is suspect. Fortunately, great effort has been expended by philosophers and other thinkers to resolve this problem; the interested reader is referred to the introduction in [16], where the overall nature of logic is discussed very clearly if

somewhat verbosely.

The notion of *proof* is also important. In logic, a proposition is proved if a valid argument can be constructed so as to demonstrate that the proposition is true. This is a well-accepted, conventional definition of the term for classical formal logic. Variants exist, depending on the form of logic used. For example, in fuzzy logic [147, 148], there are gradations of truth; thus it is possible to have propositions that are more or less true than others. However, for the current work, classical *two-valued* (i.e. *true* and *false*) logic is sufficient.

Another important notion is that of *validity*. Validity refers to the correctness of an argument, but not to the truth-hood of its premises. Thus the classical example *"If every man is mortal and Socrates is a man, then Socrates is mortal"* can be shown to be valid without making any statements about the premises *"...every man is mortal..."* and *"...Socrates is a man..."*. The advantage is that we can distinguish clearly between the premises of an argument and the procedures used to reach conclusions. Also, logic provides the means to check those procedures for incorrect intermediary steps. Although logic may also assist in determining the correctness of the premises of an argument, this is a separate consideration. Often, premises of arguments are based on observed facts, which are by definition empirical and which usually cannot be *proved* in the technical sense of the word. Although this restriction may appear to limit the applicability of logic – especially with respect to a domain with such strong physical ties as design – the notion of validity lets us prove or disprove arguments, which are the building blocks of reasoning; and reasoning is an essential component in design.

The issue of validity of particular formal systems is problematic; this was established by Gödel in his work on incompleteness. However, insofar as formal systems may be considered valid, they offer a far more rigorous means of treating phenomena such as design than any other available technique. It thus remains advantageous to employ formal techniques in design theory.

Logic is considered to be independent of the physical universe; it is for this reason that truth of statements such as *"...Socrates is a man..."* cannot be decided. Indeed, it is possible to generate formal systems that are in no way related to any aspect of physical existence. However, some formal systems have been found to be very useful in explaining and predicting the behavior of physical (extralogical) phenomena. As engineers, we are particularly interested in formal systems that do relate in some way to the physical universe. These formal systems are our *logical models* of phenomena. The success or failure of a particular

logical model depends on its perceived correspondence to the phenomenon being studied. Logicians call the correspondence between a model and an observed part of reality an *isomorphism* [149]. The more accurate and complete the isomorphism is between a model and a phenomenon, the better the model. The notion of isomorphisms will also play a role in determining the extent to which formal theories for design can be considered valid; this will be explained in the following Sections.

Furthermore, logic is, on the whole, objective. It is considered to be valid regardless of human cognitive function. It is interesting to note, however, that there does exist work in logic meant to formalize such naturally subjective domains as belief systems [150]; such work has found technological application (for example, in the treatment of distributed systems such as computer and communication networks). It would appear, then, that if we are to maintain a certain objectivity in design research, logic should be considered a very important and useful tool.

To motivate the discussions to follow, the presentation in this Part will begin with an examination of terminologic and taxonomic issues, and the inherent vagueness with which notions and concepts in design are currently defined. This view has been expressed by others in the field, especially Dixon, in [4]. Next, a distinction is drawn between the conceptual models that we use to understand design phenomena, and the computational models we use to implement our conceptual models as design aides. This distinction is important because it can significantly simplify the complexity of design theories. We then examine the concept of "self-reference" in the context of design theory. Self-reference can prevent valid formalizations in any domain; the author argues that it should be avoided in design theory if valid formalizations are to be found. Next, the notion of design as an *artificial science* is introduced as a means of discussing the role that formal logic plays in the establishment of useful frameworks within which design can be studied. The conclusions of these discussions motivate the formulation of two conceptual design theoretic tools. The first, a layered logical structure for design, outlines a technique whereby different degrees of abstraction in design information may be identified and classified. The second, called a *design space*, is meant to help study the relationships between various notions, methodologies and approaches to design.

Chapter 4

Motivating Discussion

4.1 Terminologic Considerations

There is no rigorous terminology or nomenclature for important concepts and notions in engineering design. Terms that designers and design researchers use are often defined in whatever fashion is most convenient to them (e.g. the various definitions attributed to the term *feature* in [13,51–53]). This is not an indictment of the abilities of these persons or the quality of their work, but rather an indictment of our collective ability to define the nature of design itself.

The nearest convention we have in this regard is the engineering drawing. Although the graphical nature of engineering drawings can capture some information efficiently, drawings alone, even if computerized (via CAD systems), cannot capture all the information necessary to represent the nature of a design in an efficient and usable way [32,94].

A lack of standardized nomenclature results in bad communication. Between designers, this can have disastrous consequences. In computerized systems, it can lead to inefficient, incompatible software systems that stymie rather than stimulate the abilities of designers.

At a deeper level, this indicates a significant disagreement on the limits or boundaries of various notions and concepts. What is engineering design? What is a solid model? At what point does a geometric model become a solid model? Should a finite element mesh be considered an analytic model parallel to a solid

model, a particular manifestation of a product model, or an entirely separate kind of model? To what extent are constraints a valid modeling form? Is constraint theory a modeling technique or a technique of analysis? These questions indicate just how vaguely design is defined, a notion echoed in the work of many researchers [29, 31, 39, 151]. They cannot be answered because there is consensus neither of nomenclature nor of the associated underlying concepts and notions.

The importance of this problem is underscored by its impact on our ability to teach design. In order to teach, we must communicate effectively. Without effective communication, the imprecision and inconsistencies of the teacher will tend to be passed on to the student. Thus, the problem perpetuates itself. The relationship between design theory and teaching design is discussed in some detail in [4, 10, 36]. Recognition of key primitive notions is essential to attain a more formal understanding of design. The establishment of more precise conceptual definitions and their corresponding terminologies would assist us to resolve many difficulties now being experienced.

By simple analogy, consider the nomenclature of chemical compounds. For example, a *sulfite* is different than a *sulfide*, which is different than a *sulfate*. At once, these terms concisely and exactly capture significant differences in composition and behavior of these various classes of distinct, yet related, compounds. The importance of this nomenclature is not so much that it accurately differentiates between various classes of compounds, but that it represents a collection of very precise definitions and notions that are consistent with the rest of the formal structure called chemistry.

This kind of precision is missing in design and design research. As things stand today, it is difficult – if at all possible – to generate a nomenclature of design parallel in precision to that of chemical compounds. However, if this were possible, the benefits to be reaped would be great. Of primary importance is the increased efficiency and security of information transfer. A universally recognized nomenclature would virtually eliminate the subjective interpretation of engineering information and thus greatly diminish the chances of *misinterpretation* of that information. Designers will then be able to spend more time discussing the nature of their designs and less time arguing over how the designs are presented.

Also, increased efficiency in communication can have important consequences to the development of software systems meant to assist the designer. Specifications for software systems will be more robust because the models they implement will be more precisely specified. Computers are not yet able to deal well – if at all – with vaguely defined data. Implementation details would be easier to manage if the

computational models of the problem domains included precisely defined notions.

In searching for a nomenclature for design, it will be up to design theory researchers to provide the formal systems and methods needed to assure the validity of the nomenclature. The tool they will use to provide it will have to be logic. The author's propositions in this regard are discussed in Sections 5.1 and 6.2.

4.2 Taxonomic Considerations

A taxonomy is an orderly structured system of classification based on presumed or observed properties. Taxonomies are used to classify entities and thus permit their study at higher (i.e. more general or abstract) levels than that of individual entities. Taxonomies are also very useful in systems containing many individuals and many different kinds of individuals; being able to classify individuals can be a great tool to assist in the management of information about the individuals and how they relate to each other.

This problem is nicely stated by Chignell *et al*, in [143]:

> "One of the most annoying things . . . is the feeling that one cannot keep up with this broad literature of things that one should know about. Drawing on psychological theory, it seems that the task of the researcher might be simplified somewhat by providing a framework or organizing schema within which to understand and absorb the frightening amount of possibly relevant material that should be dealt with."

One of the important aspects of taxonomies is that they need not be complete and entirely correct to provide valuable assistance to researchers. The taxonomy used for the classification of living organisms is a good example of this. Though it is not perfect (some disputes still go on as to the nature of certain organisms), it is *for the most part* a highly useful tool in such areas as the study of evolution, animal and agricultural husbandry, teaching, and so on.

Taxonomies could help design theory in many ways and at many levels. At a practical level, they could assist in standardizing parts and components, leading to universally compatible part catalogs, annotated libraries, etc. They could be used to classify design processes and so provide a framework within which designers can select appropriate methods for different kinds of design problems. Also, they could assist in the classification of manufacturing techniques, promote modular construction and thus help not only in

product manufacturing, but also in the design of the plants and assembly facilities used.

In design theory itself, taxonomies could help us understand differences and similarities in various theoretical systems, which in turn provide a means to evaluate new ideas. We could also use this knowledge to identify classes of problems that require more research, and criteria that can be used to optimize their solutions.

Taxonomies for design and design research have not yet been developed, though there have been various attempts [77, 87, 152]. In many cases, the taxonomies are derived in a generally *ad-hoc* manner [52]. A good taxonomy must be based on formal reasoning, and the principal criterion used in the search must be that of validity. The premises of taxonomic arguments are those notions for which taxonomies are sought: these notions are captured by a nomenclature. Demonstrating validity of the taxonomy does *not* include demonstrating validity of the premises.

Insofar as taxonomies are ordering mechanisms, we can look to logic to provide a number of tools to facilitate their generation. In Chapter 10, five specific mechanisms by which design information can be ordered will be discussed and demonstrated to be sound with respect to their logical foundation. These mechanisms can be taken as general principles with which taxonomies of design entities can be formed.

4.3 Computational Considerations

The author has noted that many suggested models of design inherently involve computational aspects. Examples of this are: [117], where little distinction is made between the conceptual problems of classifying design function in mechanical design and the computational issues surrounding the implementation of their classifications in computer systems; [126], wherein the factors affecting data flow in a design activity are suggested to arise both from design requirements and from the requirements of computerized implementation of their system; [77], wherein computational models of knowledge engineering are used as a basis for formal design process models; [123], where integrity of stored information is seen as an important aspect of design; and [125], where a relationship is indicated between the "business process" of computer-integrated manufacturing and the maintenance of software. It is noted that all these efforts fall at least nominally within the domain of design theory, and thus indicate a possible relationship between design theory and computer science.

The author suggests that a distinction must be made between conceptual and computational models. None of the above-noted models actually *require* implementations. Each presents a certain view of design, and contributes to our understanding of it, regardless of how the systems are implemented. The distinction between implementation of a tool and the model upon which it is based is termed by the author as the distinction between *computation* and *conceptualization*. Models can and do exist as formal theories, independent of their implementations.

In general, a conceptual model might provide formal descriptions of the kinds of information that must be present, the kinds of operations that are defined on the information, integrity and other constraints that are to be satisfied, as well as the general philosophic background. A computational model defines the *implementation* of the conceptual model, and might specify the kind of scoping[1] to be used, atomic data structures, type-checking semantics, transaction control and so forth. An appropriate computational model is based in part on the requirements of the conceptual model. Problems arising from differences between the two are often referred to as *impedance mismatches* [27]; but the problems themselves are all *computational* problems, and do not necessarily reflect the nature of the conceptual model itself.

Issues of implementation – such as the selection of a base programming language (e.g. C versus Smalltalk) – can greatly affect the success of a particular model of design. Also, computational theory can be a useful tool in design theory for its ability to formalize actions and procedures that manipulate information. But in the purely theoretic arena, any mechanism could be used, albeit awkwardly. This is due to the deterministic nature of the computer itself. In other words, the implementation does not affect the validity of the conceptual model itself.

The converse of this is also true: given a particular implementation, any valid conceptual model can be captured. To be sure, the efficiency of a particular implementation does depend on the relationship between the model and implementation techniques. However, it is noted that the term "efficiency" in this case denotes efficiency *of the implementation* and not of the model.

Therefore, the evaluation of implementations of models does little to effectively compare the underlying conceptual models, which should be evaluated on logical grounds based on their ability to explain and predict phenomena of interest. The inclusion of issues pertaining to the implementation of a formal conceptual system in a computerized environment can unnecessarily increase the system's complexity by

[1] The *scope* of a data structure is the region of program code in which it is active or accessible.

introducing a false coupling between the model and the implementation. This does nothing to increase the robustness and secure validity of the model, nor does it verify or increase the efficiency of the implementation.

It is, in fact, possible to separate issues of conceptualization from those of computation: [31,40,52] and the work presented herein all achieve this separation at least to a degree, and still make meaningful statements about design. Although these and other related efforts can be used to develop design software systems, the fact that they are not directly tied to the development of software permits them to be used for a variety of other reasons, such as teaching aides and research tools, in non-computerized arenas.

The author finds it curious that model generation for design should have become so tightly connected to the development of software systems, but believes that the connection arose from the historical roots of design theory in the development of the first CAD systems. Computers, being deterministic machines, cannot deal well with the arbitrary nature of the way design was once conducted. Hence, techniques were sought that made design more amenable as an application for the then-emergent computer technologies. Graphic rendering technology is a direct progenitor of today's solid modeling programs [121]; constructive solid geometry itself eventually led to features [5,51]. Since then, our understanding of both design and the underlying logic of computation and information theory have led to the realization that formal models are useful and important tools independent of their use in computational tasks.

4.4 Summary

In this Chapter, the author has presented a discussion intended to motivate the pursuit of a more complete, formal understanding of design. In this regard, we have examined terminologic, taxonomic and computational considerations. There is currently no consensus regarding the definition of important terms and notions that are often used in design research and practice. Without such a consensus, misinterpretation of design information and incompatibilities between subsystems cannot be avoided or even controlled. Furthermore, there exists little coordinated organizational structure for design information (taxonomies, etc.) that can streamline the specification and communication of information vital to the design endeavor. Finally, the coupling of design with computational considerations unnecessarily complicates investigations of design. In order to improve the state of our understanding of design, all these issues must be

addressed.

Chapter 5

A New View of Design

5.1 The Notion of Self-Reference

Hofstadter, in [149], states "It is very important when studying formal systems to distinguish working *within* the system from making statements or observations *about* the system." Hofstadter is writing about the concept of *self-reference*[1], and though it may seem simple enough in this short quotation, the concept is one of the most complex and consequential of this century. It is a notion fundamental to all the arguments present in the above-cited, Pulitzer Prize winning work. The problem of self-reference is defined formally by Gödel's Incompleteness Theorem, which states that no system can reference itself and be proved valid. A system without self-reference will not be able to prove some statements that are nonetheless valid; a self-referential system, on the other hand, will permit the proof of all valid statements plus certain invalid ones as well (i.e. a *paradox*). Put another way, no system that can be proved valid can be complete. No other notion of logic has had more important consequences, and yet been so universally accepted as necessary. For example, in classical set theory [153], the existence of the universal set cannot be demonstrated without appealing to self-reference; yet set theory with self-reference *and* the universal set is easily proved inconsistent (i.e. containing invalid parts). Moreover, it can be shown that any formal system that corresponds to number theory through an isomorphism is incomplete; i.e. there are some

[1] Self-reference is also known as *reflection*; however, the author prefers the former term for its direct denotation of systems that are aware of, or act upon, themselves.

truths that cannot be proved [149, 154].

Although mathematics and logic is limited in rigor due to incompleteness, the fact remains that both these disciplines have contributed inestimably to our understanding of the physical universe. There is no a-priori reason to think that similar contributions to our understanding of design are not possible.

The kind of paradox that can occur in self-referential systems is exemplified by the sentence "This sentence is false." In fact, self-reference abounds in the English language (e.g. "This is an english sentence."), indicating that english cannot be proved valid. The fact that such self-referential english sentences can be quite meaningful to humans does not bode well for the validity of the human mind. In fact, the mind itself is self-referential: how else can we think about the mind? The mental processes that we call "thinking" are part of an entity capable of self-reference, and the concern with logic of scholars through the ages is an effort to justify our thinking processes and validate the results thereof. Indeed, self-reference, or self-awareness, appears to be a property unique to the mind among all natural phenomena; it is difficult to think of gravity, DNA, or a airplane as being self-aware. While the author is not suggesting that the mind will forever elude formal understanding, we do suggest that any formal understanding of the mind will be of a different order than our understanding of other natural phenomena because of the mind's self-referential nature.

Because english, and other natural languages, are so often used in design to communicate information, self-reference can also appear in our ideas about design. A statement such as "Conceptual design is a component of the design process" is self-referential; a design process that takes into account properties of itself is self-referential. Design itself tends to be self-referential, as is evidenced by existing research: "A major part of the design activity is concerned with the development of the design process itself [101]." Any self-referential system that seeks to formalize design will be logically inconsistent. Furthermore, it is impossible to determine the extend of the inconsistency working within the system itself. If we are to find a reliable, logical system with which to model design, we must ensure that it does not contain the notion of self-reference.

The issue of self-reference is perhaps one of the greatest stumbling blocks facing design researchers, if for no other reason than that it is human nature to treat the universe in a self-referential way. However, it does seem possible to the author that beginning carefully from first principles, and striving to avoid the designers' self-referential mental processes (e.g. intuition, opinion, etc.), design can be at least *partially*

formalized into a system that is valid with respect to its logical foundation. In the Sections to follow, the author will present the beginnings of such a formal system for design.

The author has found the use of the terms *subjective* and *objective* to provide a useful viewpoint in this regard. The term *objective* is defined as existing independent of the mind, belonging to the sensible world, being observable or verifiable especially by scientific methods[2]. Insofar as design is (at least in part) a function of the human mind, there is obviously a subjective component to it. The subjective aspects of design are all prone to self-reference by their very nature. However, not all of design is subjective. Any objective component of design may be treated formally. Moreover, as research into design progresses, it may be found that aspects of design considered heretofore subjective can be treated quite objectively.

5.2 Design as an "Artificial" Science

5.2.1 The Scientific Approach

A number of researchers have suggested recently that a more "scientific" approach should be employed in the study of engineering design. Two noteworthy examples of this point of view are [11] and [31]. The argument for pursuing such a scientific approach generally proceeds as follows:

- The objective of science is to provide a precise and logical understanding of natural phenomena.
- Design as an endeavor is currently not precisely defined, and tends to be highly subjective, much as science was before the Renaissance.
- Therefore, those mechanisms that provide precision and structure for science may also be able to do so for design.

The relationship between science and design theory will be discussed in this Section. The author postulates that there is a part of design that can benefit the most from an approach based on logic rather than science. Whereas logic is seen as a necessary progenitor of both the "natural" sciences and design theory, science and design theory themselves are seen as equals related through logic. Because of the egalitarian nature of this relationship, the author introduces a new term, *artificial science*, which is intended to connote

[2]Paraphrased from Webster's 7th dictionary.

the formal nature of design theory while distinguishing it from the natural sciences. The essence of this relationship is depicted graphically in Figure 5.1.

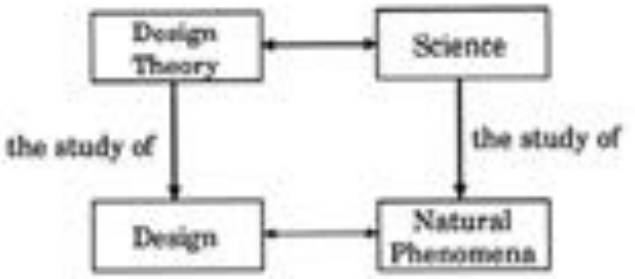


Figure 5.1: Relationship between design theory and science.

The techniques espoused by supporters of a scientific approach to design research generally seek empirical data about design (e.g. [117, 122, 126, 135]). The scientific method is used as the basic methodology in such projects: observations are made of designers at work; then, formal models are sought that can predict (at least to a degree) the behavior of designers when confronted with particular situations. Some call this kind of research *cognitive design research*, and the procedure used is in essence the same as that used by scientists to investigate natural phenomena.

Although the current literature indicates that cognitive design research has yielded many useful results, the author views such research as being targeted not at design *per se*, but rather at the mental processes of the designer, and as such tends to the subjective. This view has been espoused by at least one major proponent of new fundamental research into design ([4]). The distinction is important because statements about how designers think do not necessarily relate to design itself. We have stated earlier that mental processes occur within a self-referential system (the mind) and thus are unprovable by conventional logic techniques. However, these mental processes can be rationalized as soon as they are externalized; that is, as soon as these mental processes become manifested outside the mind – be it in the form of a CAD drawing, an english sentence, or a mathematical formula – they leave the realm of the self-referential mind and can be analyzed logically to a greater extent than if the designer's cognitive functions were included.

In this way, the author differentiates between those parts of design that cannot (currently) be formalized, namely the subjective mental processes of designers, and those that can (and should) be formalized, namely all externalizations of those processes. We note that our intention is not to remove the creative, cognitive components from design, but rather to provide the means by which to analyze the results of these

processes in a logical manner, thus helping the designer to *channel* his/her imagination and creativity in directions that will maximize results. Many of the best mathematicians and scientists throughout history have been very creative and intuitive people, and have used these characteristics in their work to great advantage. This should be the case in design as well. Also, formalization of the non-mental segments of design can be beneficial to cognitive design researchers by providing them with a yard-stick against which to make observations and compare theories.

5.2.2 Design Versus Natural Phenomena

The *scientific method* is a trial and error technique, the goal of which is the creation of models based on observations that let us understand a given phenomenon. The logical and mathematical models created by scientists have provided excellent isomorphisms to various natural phenomena. Based on these successes, it could be argued that logic in some way reflects an essential property of the universe. However, this proposal begs the question of whether the universe is deterministic [16]. In order to avoid this contentious and rather philosophical issue here, the author adopts a somewhat less aggressive position: that formal models *approximate* some underlying structure of the observed phenomenon.

An important assumption is built into the scientific approach to design theory; namely, that there is a correspondence between design and natural phenomena, and that this correspondence allows researchers to treat design as a natural phenomenon. In other words, the same isomorphisms are relevant to both design and natural phenomena. The author contends that this assumption is misleading; we make our case with the following argument:

Design, unlike natural phenomena, is "contrived" in that it is a purely human invention. While nature may be considered as existing without any action on the part of humans, design is not independent of human beings; in fact, the designer is the only agent by which design is manifested at all [14].

Also, the evolution of design has proceeded over the years in a more or less *ad-hoc* manner, responding not only to the emergence of new scientific and technological understanding, but also to various sociological, economic and governmental pressures, none of which can be said to be particularly natural (in the scientific sense of the word).

It may be argued that since the human mind is a natural phenomenon, processes that occur within the

mind (such as design) should be considered likewise. The author argues against this position. There is no dependence of natural phenomena on human beings. However, design *requires* the existence of the mind. This distinguishes design from natural phenomena.

Science has had little success in understanding the human mind so far [155–157], much less success than it has had in understanding natural phenomena such as gravity, nuclear reactions, and DNA. One possible explanation is that the isomorphisms that have successfully been applied to natural phenomena are not accurate with respect to the human mind, and because design is a construct of the mind, we may say the same about design. If we cannot understand the mind, how can we understand design, which is an invention of the mind? The author is forced to conclude, then, that it is inappropriate to treat design as a natural phenomenon that can be studied scientifically, because it is a mental process.

5.2.3 The Role of Design Theory

Having apparently cornered ourselves in this way, we are left asking whether any formalization of design is possible at all. The solution to this quandary lies in recognizing that we do not *need* the same isomorphisms to apply to design as apply to natural phenomena. Because design is a mental process, it can benefit from the same logical thinking that permits scientists, mathematicians and logicians to solve problems more complex than they could if they had *only* their intuition and creativity to guide them. But because design is not bound by the structure of nature, we are free to make of it whatever we choose.

Though relatively unconstrained by nature, design theorists should nonetheless seek as formal and objective a definition of design as possible. We are free to do so without being constrained by the influence of science because design is not a natural phenomenon. No formal system is related de-facto to reality; it is the discovery of isomorphisms between the formal system and reality that makes it relevant. Any system for which an isomorphism to a phenomenon can be found becomes a candidate model that can be used within the scientific method. The use of logic is required because it is the only tool mankind has devised so far to reason in a reliable and repeatable way. Design theory should thus depend on logic, but not on science, for rigor. Hence, the author views design theory as a sibling, or equal, of the natural sciences, sharing with them a dependency on logic (see Figure 5.1).

This is not to say that design is not related to nature at all. Usually, the ultimate result of design is an

artifact having some physical existence; thus it must relate to nature in some way. But this is what design *does*; here we are concerned with what it *is*.

In summary, the author contends that design theory should be concerned with finding logical systems that can formalize design without necessarily relying on the isomorphisms of the natural sciences. In an effort to emphasize this notion, the author introduces the term *artificial science* to describe design theory. This term has been chosen to distinguish clearly between design *theory* and the natural *sciences* while preserving the idea of some commonality between *design* and natural *phenomena*.

The author notes that there is already a tendency to consider design in some way artificial; for example, the term *synthesis* is often used to describe the methodical generation of a design artifact, even in conventional design contexts (e.g. [34]). While *synthesis* is often taken roughly to mean *creation*, it has a connotation of artificiality that is generally missing in contexts where *creation* is used[3].

Since design is something that begins in the human mind but ends in the real world, there are some aspects of design that logic cannot be expected to capture on empirical grounds. In addition to creativity, intuition and opinion, "facts" from the real world cannot be dealt with using logic alone, in the same way as the atomic premises of the syllogism about Socrates in Chapter 3 cannot be dealt with using logic alone; here is another region where natural phenomena influence design, and where the conventional scientific method may be used. Still, there remain many aspects of design that are candidates for formalization through logic.

To this end, the author postulates that a logical system to model engineering design can be achieved. The system would be used to represent facts and to reason about design. Theories about design and design information may be derived within the model and eventually supported or disproved by logical analysis, experimentation (i.e. application of the theory to test situations), and observation of the resulting systems.

The derivation of a logical system for design as an artificial science is the principal goal of the author's work. The first concern is to identify tools of logic that provide good isomorphisms. In seeking such a system, a return to first principles has been found necessary to limit empiricism, self-reference and the influence of the designer's mental processes. The result of the author's efforts in this regard is presented in Part III.

[3]Based on definitions for *synthesis* and *creation* in the Oxford English Dictionary.

5.3 Summary

This Chapter has discussed the advantages and problems associated with the use of formal systems in design theory. While they represent the best understanding mankind has of logical, structured argumentation, formal systems are inherently limited by Gödel's Incompleteness Theorem. Nonetheless, no superior approach to systematic, formal reasoning exists, so in order to maximize the degree of rigor in any attempted formalization of design information, formal systems should be considered an essential tool.

Furthermore, the author introduces the term *artificial science* to describe design theory as a sibling of the natural sciences, sharing with them a dependence on logic for formal rigor. It is unreasonable to expect our understanding of natural phenomena (the domain of the natural sciences) to contribute to our understanding of design, because design is manifested as a construct of the human mind rather than being a natural phenomenon independent of human cognition. Although we lack a good understanding of the human mind, the *externalizations* of our thought processes can, and should, be subjected to logical analysis. Such analysis can identify inconsistencies that might otherwise escape detection. Furthermore, the formal techniques of logic can help a designer channel his/her creative and intuitive energies in directions more likely to lead to successful design solutions.

Chapter 6

Logical Solutions

6.1 Limiting Self-Reference in Design

In preceding Sections, it has been suggested that it is possible to construct a valid formal system for those parts of design that are independent of human mental function. Such a system should be able to deal not only with specific information regarding a particular design artifact, but also with various degrees of abstract information that are equally essential to the design endeavor. However, we must impose a certain structure upon the system to avoid self-reference. The structure is such that the total system is composed of logical sub-systems of increasing degree of abstraction. Each sub-system is capable of referring to other, lower layers, but not to itself, or to higher (more abstract) layers. This is in essence the solution suggested by Bertrand Russell to a large class of paradoxes in the original derivations of classical set theory [153] that arose due to self-reference. The resulting layered structure consists of one layer for each degree of abstraction. In the general case, where the domain of a logical system includes the entire universe, an infinite number of layers would be needed to capture all possible abstractions. Fortunately, due to the relatively restricted domain of design (with respect to the general case), and because each layer would have a distinct meaning in design (via the isomorphism), a layered system of logic should be tractable.

In this Section, the author presents the beginnings of such a layered system. The presentation is necessarily

quite general, but it does suggest the overall structure of the system, and indicates how the various degrees of abstraction are delimited based on the notion of avoiding self-reference[1]. The structure is depicted graphically in Figure 6.1. The relationship between the Hybrid Model (HM) and the rest of the layered structure is also shown in the figure. HM is presented in Part III.

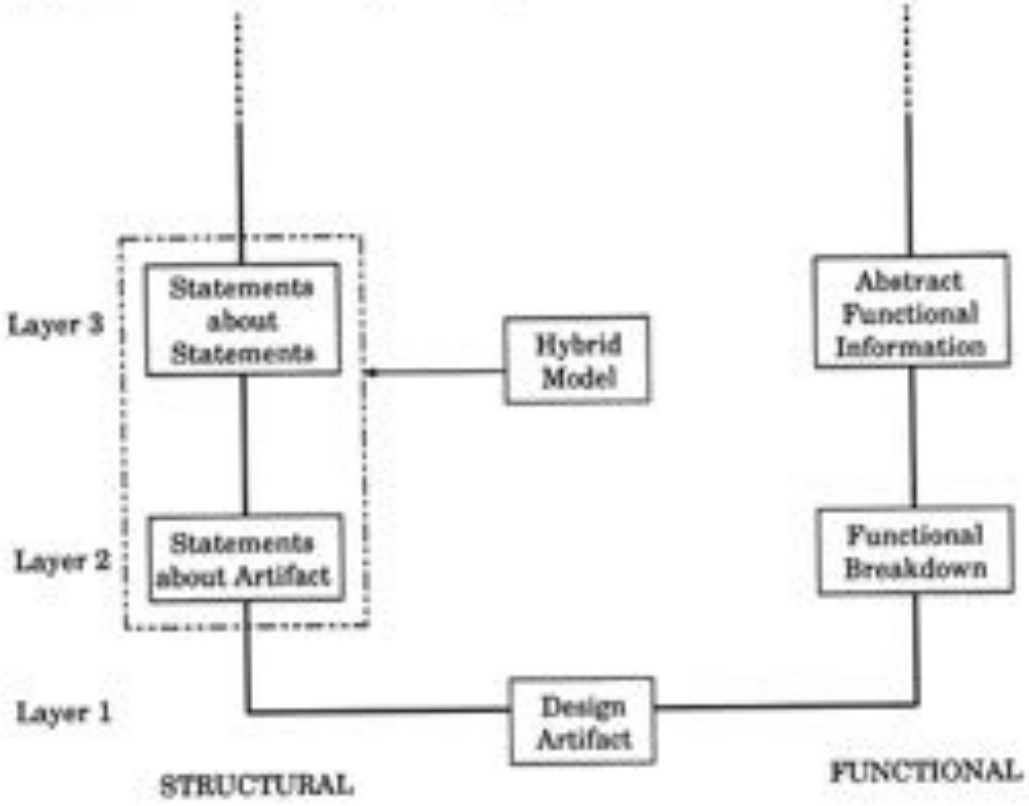


Figure 6.1: Logical structure for removal of self-reference

At the lowest (least abstract) level of the system, is the design artifact itself; this is the actual part/object that is the goal of a design process, an actual physical entity. We immediately divide the system into two different but complementary branches. On the one hand, there are all the statements that can be made about the artifact at any instant in its existence (or even before or during its creation); on the other hand, there are all the actions that are required to create the artifact, to use it, to maintain it, etc. The separation is essentially one of *structure* (description) versus *function* (procedure). The branches are complementary in that they both relate to the design artifact. The structural branch is static, time-independent and prescriptive, while the functional branch is dynamic, time-dependent and descriptive. Furthermore, the

[1] It is interesting to note that though the system described in this Section is not self-referential, the description itself presented herein depends on self-reference to achieve its end.

structural branch addresses the issue of *what* design is, whereas the functional branch deals with the *how* of design's occurrence and procedure.

At higher levels of abstraction, the functional branch captures the design process used to create the artifact and issues of cognition by the designer. Thus, the functional branch may exhibit self-reference. The structural branch exhibits no self-reference because it does not consider the role of the designer. Thus in this framework, we can *identify* and *limit* the effects of self-reference.

The structural branch captures the state of information pertaining to a description of the artifact at various degrees of abstraction. Since this description is independent of the actions that caused the information to be discovered or generated, it is applicable uniformly throughout the design process. Also, since the description is independent of the agents responsible for those actions (the designers), it is also applicable to any design process; that is, the information description is independent of the design process. Thus, it is possible to control self-reference in the structural branch. HM, which is the heart of the author's work, is contained entirely in the structural branch.

Each layer in the proposed system makes statements about the preceding layer; this is the case in both branches. Thus, the first layer beyond the design artifact in the structural branch contains statements about the artifact, capturing concrete facts about it. The second layer contains statements regarding the classification of these facts, and by extension, the classification of different artifacts. The next layer (not shown in the figure) contains statements about different classification schemes and mechanisms.

Similarly, in the functional branch, the first layer beyond the design artifact contains functions provided by the artifact and a functional breakdown of the artifact. The second layer contains actions taken to create (design *or* manufacture) the artifact. These actions may be affected by the cognitive processes of the designer. The next higher level (not shown) contains more abstract information about those instructions. Classification of actions would also occur in this layer, and can include judgmental and intuitive remarks about the relative merit of those instructions. The next level (also not shown) would include statements used to reason about the classifications and would include issues of decision making and negotiation between designers. This information would be used in the generation of different design methodologies, their analysis and comparison.

The process of abstraction is used in both branches to move from one layer to the next. Clearly, the abstraction could continue *ad infinitum*, generating innumerable layers. However, the kinds of statements

captured by these layers would quickly become so abstract as to be entirely meaningless. At this time, the author believes that three or four layers should be sufficient to capture all relevant information. It is also noted that this layered structure can be useful in the generation of design taxonomies by providing a criterion for separating statements (i.e. information) based on the degree of abstraction used.

6.2 Categorization of Design Aspects

In response to the lack of formal terminology and taxonomy for design and design theory, the author proposes a mechanism that can assist in the organization of relevant notions and their corresponding terms. The mechanism proposed herein arose from the author's consideration of many aspects of design, only one of which is mentioned here, by way of an example.

Keirouz et al [122] discuss the differences between variational and parametric modeling from the point of view of constraint satisfaction in conceptual design. The current author considered how best to categorize the cited work: it did not belong strictly in any one of the three areas mentioned (modeling, constraint theory and conceptual design) but seemed to relate to the three together.

This kind of tightly interwoven dependency between various aspects of design is indicative of the high degree of complexity required to accurately model it. Consideration of each of the aspects alone is not sufficient because a large part of the complexity arises from the relationships that exist between them. The author therefore sought some mechanism that could represent the various aspects of design as individual components while also capturing the relationships that exist between them. The mechanism is intended primarily as a conceptual tool, an aid to stimulate clear thinking about a potentially confusing problem.

The author's research suggested that in a real design process, there are a number of different, fairly independent aspects that interrelate. Due to the richness and complexity of these relationships, a multi-dimensional approach seemed appropriate.

The author thus proposes the use of a *design space* composed of orthogonal axes. Each axis represents an independent aspect of design. In this system, different relationships, approaches and techniques can be classified and compared. A particular relationship can be represented as a point, line or region in the design space. The author has identified four orthogonal aspects of design: artifact modeling (the λ-axis),

behavioral modeling (the $\mathcal{B}$-axis), meta-modeling (the $\mathcal{M}$-axis), and model implementations (the $\mathcal{I}$-axis). Each aspect is assigned to an axis in the space, each of which is discussed briefly below. Figure 6.2 represents the four dimensions of the design space as two three-dimensional spaces sharing two axes, and includes a shaded area representative of the "locus" of the work described in [122].

6.2.1 Artifact Modeling

The ultimate goal of a design process is the production of an artifact or product. Thus, one aspect of design is the ability to state in precise terms the nature of the artifact, that is, the generation of a *model* of the artifact itself. Information from this model is eventually used in a number of other areas (analysis, manufacturing, etc.).

For this reason, one axis in the design space is allocated for design artifact modeling techniques, including the physical aspects and the physical relationships between components in the artifact. Variational and parametric modeling, as well as various solid modeling techniques, would all be represented on this axis. In Figure 6.2, artifact modeling is represented by the $\mathcal{A}$-axis.

It is interesting to consider the role of constraints in artifact modeling. Are constraints, vis-a-vis constraint satisfaction, a *necessary* part of artifact modeling? Or are they essentially orthogonal to artifact modeling?

There are ongoing efforts in the field to investigate "constraint-based design" from points of view ranging from knowledge-based systems [23, 55] to alternate parameterization schemes [72, 158] and constraint-based design [67]. These efforts have all had at least some success in embedding constraints into other design aspects. However, constraint theory is in the most pragmatic sense an analytic technique and not a modeling technique: it permits the mathematical study of the capability of an artifact to provide a given functionality [63] (this notion is discussed in more detail in the next Section). The author recognizes that constraints can also be used to analyze and study modeling techniques; but in such cases, the constraints apply to the model, not to the artifact being modeled, and so must be regarded separately. The design space being discussed in this Section is meant to study design; hence, constraints are not involved in the artifact modeling axis.

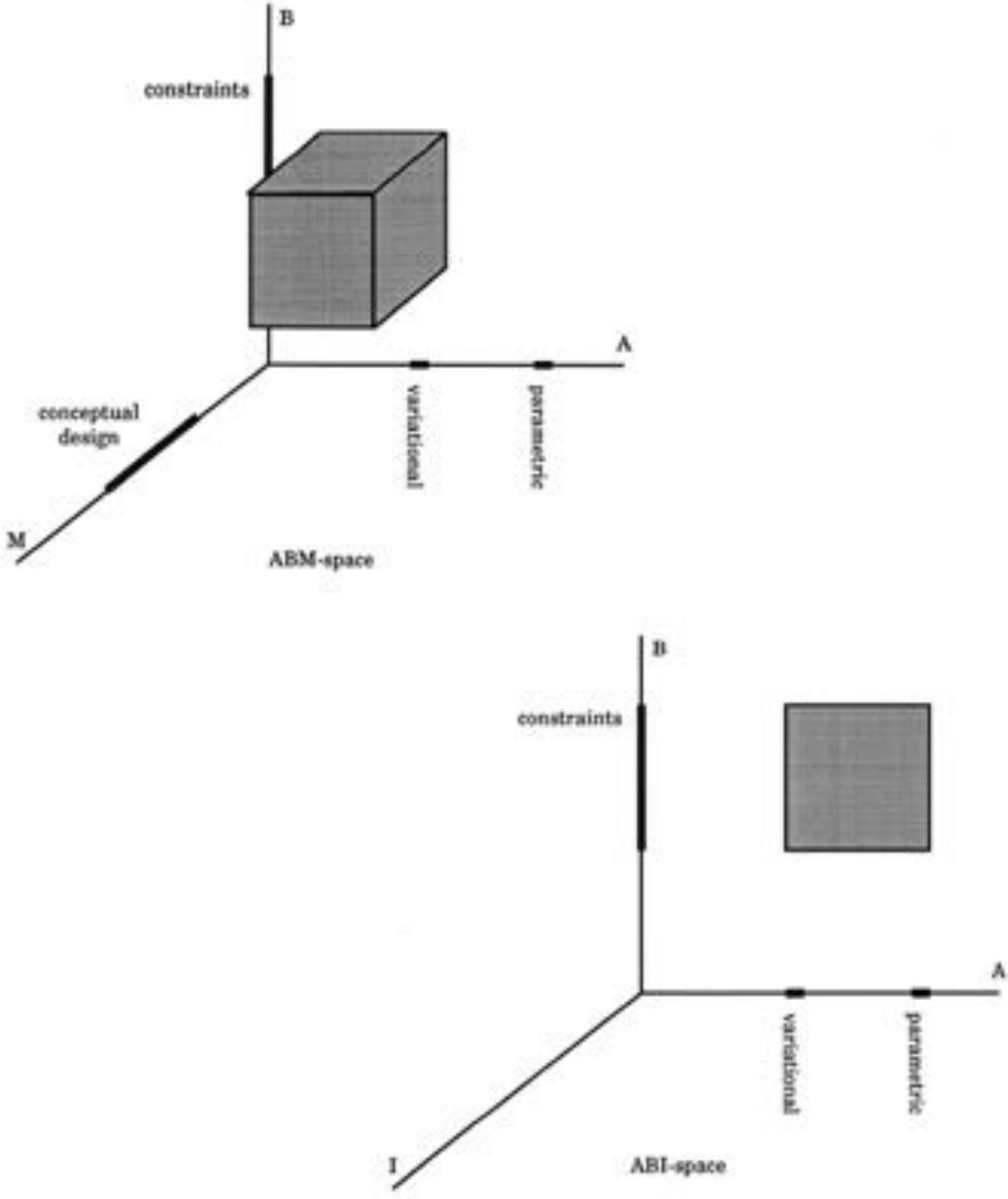


Figure 6.2: Graphical representation of the 4D Design Space as two 3D spaces

6.2.2 Behavioral Modeling

Behavioral modeling captures the other side of the artifact modeling coin. This aspect of design is concerned with modeling the *response* of the artifact to stimuli provided to it from its operating environment.

Constraint satisfaction is a primary technique with which to model system behavior. In many cases, a design problem begins with the specification of some objective function to be met by a design. This objective function is derived from issues concerning the environment within which the design is to function, and takes the form of a constraint. Other constraints, including those internal to the artifact itself, are usually derived in some fashion from the objective function.

Constraints permit the creation of mathematical models of artifact behavior. Mathematical modeling of systems (e.g. finite element or kinematic analysis) is therefore also included in this aspect of design. Optimization and simulation based on the mathematical representation of behavior are also included.

Constraints capture the relationships between form and function, and also between the artifact and the environment within which the artifact is to function. They thus represent the link between an artifact's structural model, which is usually represented as being isolated from its environment, and the rest of the world in which it is intended to function.

A second axis of the design space is used to represent behavioral modeling, and is labelled the B-axis in Figure 6.2.

6.2.3 Design Evolution, or Meta-Modeling

Such notions as conceptual design, detailed design, concurrent design, etc. do not relate directly to the artifact itself, but rather to the system by which the artifact is produced. As discussed in Section 5.1, such notions exist at a different level of abstraction than those of artifact and behavior modeling. By treating more abstract notions separately, we can eliminate a possible source of self-reference in our framework. Conceptual, detailed, strategic, and other "kinds" of design are *meta*-modeling techniques that permit the study of the models themselves, rather than of the thing that is modeled (the design artifact).

Thus, a third axis is needed to represent these meta-modeling notions of design theory. In Figure 6.2, meta-modeling information is represented by the M-axis.

6.2.4 Implementations

In Section 4.3, we indicated that a distinct separation should exist between models of designs and design processes, and how we implement those models. It is appropriate then to propose a fourth axis in the design space to represent the implementations of the aspects of design represented by the other axes. Computer implementations of formal models is one component of this axis, but implementations need not be dependent on the use of computers. Whenever an idea, formalized or not, is implemented in any way, a number of other issues that are implementation-specific – issues that are not important at the modeling level – must be considered. All these implementation issues and the techniques we use to handle them are represented along the fourth axis of the design space. We have labelled the last axis as the I-axis in Figure 6.2.

Insofar as computers are concerned, all issues regarding software design, testing and usage would be represented on the implementation axis; these include matters regarding databases, computer languages, graphics, etc.

6.2.5 The Example, Revisited

By way of example, Figure 6.2 shows the regions occupied by the work in [122], mentioned at the beginning of this Section. We may quantify the cited work as a volume in the design space bounded by the design aspects marked on the axes: parametric and variational modeling, conceptual design, and constraint satisfaction (also labelled in Figure 6.2); and since the work does not discuss issues of implementation, it appears as a two-dimensional region in the ABI-space.

The quantification that is possible through the use of the design space can permit a new degree of organization in the work of design researchers. Its graphical representation permits easy visualization of the relationships that exist between different research efforts. It could be used to organize individual research projects within large groups and indicate regions where more work is needed or where different projects overlap. It may also find use in the organization of engineering corporations and controlling/regulatory bodies by clearly marking the boundaries of the areas of influence of each body. The design space can even be used to organize conferences and other meetings by permitting visual identification of areas covered by each presented work or representative group.

6.2.6 Relationship to the Layered Structure

There is a relationship between the layered structure in Figure 6.1 and the design space in Figure 6.2. The layered structure separates degrees of abstraction in design, whereas the design space separates orthogonal aspects of design. The A (artifact modeling) and B (behavioral modeling) axes of the design space represent the structural and functional branches of the layered structure respectively. Also, the M-axis (meta-modeling) captures the increasing degree of abstraction that occurs through the layered structure. Since the layered structure does not deal with implementation issues, there is no correspondence with the I-axis.

Therefore, both the layered structure and the design space capture the same basic philosophical notions, albeit from different points of view.

6.3 Summary

The author has suggested two possible solutions for addressing problems in design and design theory caused by logical inconsistency. The first is a logical layered structure that permits the clear distinction of different degrees of abstraction (Section 6.1). Being able to classify statements made about design according to their degree of abstraction, we may better avoid circular and self-referential arguments that cannot be validated. In other words, it contributes to clearer thinking about design.

Secondly, the design space described in Section 6.2 permits the visualization of the relationships offered by various approaches and techniques in design and design theory along four orthogonal (independent) metrics: artifact modeling, behavioral modeling, meta-modeling, and model implementation. Again, the principal goal is to *clarify* the relationships inherent in our understanding of design so that we may study and improve that understanding.

Chapter 7

Discussion

This Part of the author's work has covered a fairly wide range of topics, but the underlying philosophic notions are few and distinct. The goal of this summary is to consolidate the matters presented above.

The author has identified some problems in design and design theory that arise from the lack of a formal base upon which rigorous understanding can be developed. The changing, growing nature of design is in part responsible for the lack of accurate taxonomies of design components, notions, methodologies, etc. The fluid nature of design's development makes finding a correct and meaningful taxonomy quite like hitting a moving target. The non-logical *ad-hoc* nature of the evolution of design is also to blame. Its relatively arbitrary development has led to the introduction into common usage of vaguely and/or inconsistently defined concepts and terms. Ill-defined classification systems for design artifacts, components, systems, etc. have hampered the generation of appropriate tools with which design can be studied in an abstract sense. Subjectivity introduced by considerations of the designer's role in design has introduced self-reference, which in turn leads to inconsistent design theories. These shortcomings have harmed our ability to communicate and have spawned incompatibilities between the various aspects of design, leading to so-called "islands of automation". The inability to communicate properly has also affected our ability to teach design, and thus perpetuates the inconsistencies.

Empirical studies are unlikely to lead to a more scientific understanding of design because the influence of self-reference within such frameworks cannot be dealt with. Of course, existing design theories and methodologies can be valuable in guiding our search for a logical design theory, but we should not be

surprised when inconsistencies in our current understanding are found.

Self-reference can be minimized only by forming logical systems for design that do not include the subjective, cognitive functions of the designer. Logic offers us many techniques that can be used in this regard. These techniques must be diligently applied throughout our efforts, much as they have been in the formalization of science.

Theories of design can be regarded as being either functional or structural in nature. Functional theories describe artifacts from the point of view of the functions they provide and the actions needed to create them. These seem most prone to self-reference because the actions can be traced to the cognitive functions of the designer. Structural approaches can result in formal theories without self-reference; such theories take the form of prescriptive informative descriptions of design artifacts at various degrees of abstraction.

Some may argue that the intensive use of logic in design can stifle such intangible and qualitative things as creativity, opinion, intuition and judgment. The author maintains that this is not the case. Logic does not squelch creativity and intuition, but rather channels it, helping to keep the designer from making errors that would adversely affect productivity and efficiency. It helps a person have more informed opinions and make more educated judgments.

Three informal, conceptual tools have been introduced. These tools are meant to clarify and study the postulated logical structure of design. The notion of design as an *artificial science* is presented to reconcile the differences between the "natural" sciences and design theory, and to present a point of view that permits design researchers to take advantage of the formal tools of logic more fully. The *layered logical structure* presented in Section 6.1 permits the modularization of design by degrees of abstraction. It allows for the identification and subsequent elimination of some occurrences of self-reference from design theory, and may be useful in the generation of design taxonomies. Finally, the notion of a *design space* permits the classification of the various techniques available to designers, and assists in the organization of the efforts of design researchers and theorists.

The intention in this Part has been to introduce in a relatively informal but detailed way the domain of the author's work. Having established this definitional framework, we may now proceed to detailed considerations regarding the structuring of design information.

Part III

FORMALIZING DESIGN INFORMATION

Chapter 8

Introduction

This Part of the author's work is devoted to an examination of design information with the aim of formalizing its structure. This is done by proposing a formal system that relies on axiomatic set theory for internal consistency. The author calls the resulting model the *Hybrid Model* (HM) of design information. The origin of the name "Hybrid model" is in the author's original research in combining object-oriented data models [159] with hypertext [160], hence the use of "hybrid". Since then, HM has evolved into an entirely different kind of model.

In keeping with the observations made in Section 6.1 regarding the limiting of self-reference through the use of logical layers of increasing abstraction, we will present HM in two Chapters. First, matters relating to actual design information will be dealt with in Chapter 9; then, the organization of that information will be dealt with in Chapter 10.

As well, it was indicated in Section 6.1 that a distinct separation between structural and functional descriptions of design artifacts lends itself well to the control of self-reference, an instinctively human action which has been argued to be detrimental to the development of rigorous design theories. This implies a separation between design information and processes that manipulate or otherwise use that information. It has already been suggested [7, 161, 162] that design information can be considered separate from the engineering design process.

Finally, the author contends that a good understanding of design information must *precede* any real

understanding of the design process itself. This issue will be discussed in Section 8.1, and motivates the central thesis of this work: the search for a formal theory of design information.

8.1 Modeling the Design Process

In this Section, the author presents a model of the design process with the intention of supporting the hypothesis regarding the separation of design information and processes. It is presented only to the extent that it provides a reference point for the development of HM, and brings to light several important aspects of the design process that have affected the development of HM. The model regards the design process from a functional point of view, and we refer to it simply as the *functional model* (not to be confused with the functional branch in figure 6.1).

We begin by making the relatively trivial statement of a generic mathematical function, namely:

$$y = f(x).$$

Here, f is some function that maps an input value represented by the variable x to some output value represented by the variable y; indeed, y and $f(x)$ are identical. Now, from the point of view of design, we rewrite this equation as:

$$S = d(P).$$

P represents a design problem, S its solution and d the design process. We may state this in words as: "*There is a design process that operates on a particular design problem and results in a corresponding design as a solution*". This is not an unreasonable statement to make, and though it may still appear trivial, it does carry some important implications:

Clearly, the solution depends on the problem (the output is the dependent variable). Also, as stated above, S and $d(P)$ are identical.

From a purely mathematical point of view, one may be inclined to stop here. But there is more than one

way to design a certain entity. That is, given a particular design problem, there may be more than one design process d that can provide equally acceptable solutions. Selection of a design process depends at least in part on the kinds of information about the problem that are available to the designer. (It would also depend on optimization considerations, as well as on the more subjective preferences and judgments of the designer.)

If the input is badly or incorrectly defined or specified, then selection of an appropriate solution function may be difficult or impossible. That is, if the design problem is badly stated, or is based on faulty vaguely or verbosely presented information, then the selection of a design process is prone to error (since, as is indicated above, the design process is dependent on the problem), and thus reduces confidence in the solution. Therefore, the problem must be *clearly understood* and *precisely defined before* a solution process can be selected and applied.

There is another issue that is an essential component of almost every non-trivial design task: iteration. That is, a (possibly dynamically changing) design process will be applied iteratively to a design problem in order to reach a final solution. We can represent this in our mathematical notation by:

$$S_{i+1} = d(S_i + P).$$

For each iteration $i + 1$, the design process d is applied to the problem *plus* the solution, such as it exists, at iteration i. Put another way, the solution at iteration $i + 1$ is based on both the problem *and* the ith solution. Without including the solution at iteration i in the argument to the design function, convergence would never occur. So, at each iteration in a design cycle, the existing – though possibly incomplete and/or incorrect – solution is used to drive the next iteration of the design cycle. The essential observation here is the "superposition" of the problem with the ith solution: correspondingly, the problem and the solution must be representable in a compatible way or the iteration process cannot proceed.

In summary, the functional model provides two important insights into the requirements that must be met by a formal system for design information:

- a formal understanding of design state information is necessary *before* the design process can be successfully formalized to any significant degree (i.e. design state information is independent of the

design process);

- the organization of information is relevant both for the problem definition and the solution, and *any theory of this information must be* unified *over both problem and solution domains*.

This approach differs from that taken by other researchers. Suh [31] states design problems in terms of functional requirements to be met by successful candidate designs, and solutions are stated in terms of parametric representations of variables. Another approach is taken in by Yoshikawa [39], who defines two separate spaces – a function space and an attribute space – and each of the design problem and solution are defined in terms of one of these spaces only. These approaches, among others, are similar in that both consider a design artifact (as represented by Suh's design parameters or Yoshikawa's attribute space) as a separate entity from the problem that caused it to be designed (the functional requirements or space).

This dual representation of design information by functions on the one hand and parameters on the other makes unified representation of design problems and solutions much more difficult. As well, it introduces coupling between the form of the representation of information and the design processes that use this information. In its favor, such a separation of the domains of functional requirements and physical parameters is beneficial from a conceptual point of view, permitting modularization of the task into smaller segments that can be studied individually. However, it can also lead to a divergence at the theoretic level that will prevent final integration of these domains into a single, global theory. Also, it does not address the dependence of iteration on the successful combination of information regarding both problems and partial solutions.

The approach presented herein by the functional model is superior because it simplifies the management and organization of design as an endeavor. In an iterative process such as design, the *cumulative* information generated from the iteration is an essential component of finding a correct solution. In order to merge the accumulated information with the design problem for the iteration to continue, a *unified* representation of both problem and solution must exist. The functional model of design maintains the integrity of problem and solution specification while dividing the problem along a different and more important boundary between static, passive information and dynamic, active functions that *transform* the information.

The author also notes that a number of other researchers have supported the notion of separating rep-

resentation of information from processes affecting that information, including [14,41,42,87,163]. Of particular interest is Fauvel's work [41], which suggests that the relation

```
(Activity (n), Embodiment (n)) → Activity (n+1)
```

is representative of the design process. An `Activity (i)` is some component process of the overall design process and an `Embodiment (i)` is the physical manifestation of the result of the completion of an activity. It is interesting to note the shift in point of view between the author's model and that of Fauvel. The latter is based on the notion that given some initial design activity, the results of that activity drive the selection and execution of other activities. The former is based on the notion that an initial embodiment (in Fauvel's terms) drives the selection and execution of activities that lead to other embodiments.

The author maintains the functional model as presented above because of the observation that reliable, accurate and usable information must exist *prior* to the selection of any processes meant to act on this information; that is, the emphasis should be placed on information as the driving force behind a design enterprise.

Fauvel reasons in detail on the role of various kinds of activities that are relevant to design, without dwelling on the nature of the embodiments. His results are quite clean and elegant; this encourages the author to believe that the separation of design information from design actions is not only appropriate, but necessary if design theory is ever to meet with success.

8.2 Basic Structures and Concepts

8.2.1 Basic Aim of HM

8.2.1.1 A Prescriptive, Axiomatic Approach

The aim of HM is to provide a prescriptive, axiomatic theory of the information present during the course of a design. Naturally, only information relevant to a particular design task is considered, thus restricting the application domain significantly. This restriction plays an important role in the development of HM; this is discussed below.

HM is *prescriptive* in that it prescribes a language for the specification of information for a design task. The term is not used to prescribe methodologies that should be considered norms (sometimes called a *normative* approach). Given the role of human cognition in design, the author believes that the best results can be achieved by a *symbiotic* relationship between the designer's innate capabilities (including such intangibles as judgment, creativity and intuition) and some more formal logical system.

The term *prescriptive* is used to indicate that HM is a system that lies outside the actual thought processes of the designer; that is, it lies within an objective, logical domain. The author considers this a more reasonable objective than that of the *descriptive* school [11], which seeks to quantify and formalize the actual cognitive functions (i.e. mental content and processes) of the designer.

HM is *axiomatic* in that it relies on axiomatic set theory as its foundation. The author's initial attempts sought a formalization based on existing information management paradigms (object orientation and hypertext in particular), but the lack of existing formalization in these fields was found to be insurmountable. Object orientation is often referred to more as a philosophy or point of view than an actual formal paradigm [27, 164]; the status of hypertext is even more tenuous [143, 160]. It became necessary to return to more basic first principles, and it was during the author's study of symbolic logic that axiomatic set theory presented the necessary isomorphisms upon which to base HM.

Axiomatic set theory has taken on various forms [1, 16, 153], but every form is based on the classical theory developed by Zermelo and Fraenkel [80] and which is generally referred to as ZF set theory, or just ZF. This convention is adopted in the sequel, for brevity's sake. ZF requires only the predicate calculus and is thus derived purely from logic, without any extra-logical or other empirical influences.

ZF deals with groups of completely general entities; a group of entities is called a *set*. The theory formalizes the nature of sets to such a degree as to permit the derivation of almost all the classical branches of mathematics and logic, including arithmetic, algebra and calculus [153]. The most interesting implication of set theory as far as the author is concerned regards consistency of theories that are supersets of classical axiomatic set theory. In [80], it is proved that any axiom system that can be rewritten in terms of ZF without introducing any new atomic statements, quantifiers or connectives, is consistent (insofar as ZF is consistent). In ZF, the primitives are $=$ and $\in$; connectives are binary operators such as $\cap$ and $\cup$; and the quantifiers are $\forall$ and $\exists$. As will be seen, this consistency criterion is satisfied by HM. This means that we know at once that HM is no less consistent than ZF.

8.2.1.2 Universe of Discourse and Design Entities

The term *universe of discourse* denotes the overall domain within which all interesting arguments are made. The entities contained within the universe of discourse comprise a complete vocabulary.

In this work, the universe of discourse is that of *design information*; that is, the specification of – or statement of facts about – a design problem and the various components and aspects of its solution, without consideration to any processes required to generate that solution. Within this universe of discourse, the entities of relevance are whatever design entities are available to the designer in order that he/she may fulfill the task at hand. This greatly restricts the space of possible entities (as compared to, for example, ZF, where any item at all may be considered to fall within the universe of discourse). It is exactly because of the specific nature of the entities involved that much more can be said about them than is normally possible (as, again for example, in ZF). *The notion of a restricted universe of discourse is essential in order to be able to derive HM at all.*

The author informally defines a *design entity* in HM as some real-world structure that is meaningful from a design point of view. This unit need not be physically realizable *per se*: it can be a purely conceptual item, such as a run of a finite element program or a manufacturing process plan. It can also be a *feature*, in that threads, holes, fillets, etc. are also design entities. HM deals, then, with the formalization of design entities. In the Sections following, the exact formalization is stated and discussed.

8.2.2 Theory of Logical Types and Set Theory

In ZF, if a universe of discourse consisting of all the possible sets is considered, it is very easy to generate a number of paradoxes that cause the theory to become inconsistent [80, 153]. A number of schemes have been suggested over the years to avoid these paradoxes. Two of the classical approaches are the type-theoretic approach, and the approach of class-inclusion.

Class inclusion assumes a universe of discourse containing both sets and "classes", the latter being collections of sets (not sets of sets). The resulting theories are quite powerful, but tend to hide some of the features of set theory that the author considers important for design. The type-theoretic approach, on the other hand, tends to be more explicit, but more complicated to manage as well.

In a type-theoretic set theory, logical paradoxes are avoided by restricting the kinds of individuals that can exist in various sets. At the lowest level in the type-theoretic approach exist the individuals in the universe of discourse and the attributes that can be predicated on them (i.e. acted on by functions). At the next higher level exist sets (collections of individuals) and the attributes that can act on these sets and on individuals. At the 3rd level exist sets of sets and the attributes that are predicable on sets of sets and entities at lower levels; and so forth. There are an infinite number of levels.

Furthermore, equations in a type-theoretic system cannot mix predicates from different levels *ad-hoc*, but only according to the following rules:

$$x^i = y^i \tag{8.1}$$

$$x^i \in y^{i+1} \tag{8.2}$$

where i represents the level of an entity. The first rule states that entities that are *equal* must exist at the same logical level; i.e. they must be, for example, both sets, or both sets of sets. The second rule states that if one item (x) is a member of another (y), then the former must be one logical level lower than the latter. Thus, if we consider that (a) x is a set, and (b) that $x \in y$, then the rules of type theory tell us that y *must* belong to the level containing sets of sets (since x is itself a set).

The problem with the type-theoretic approach is that the bookkeeping required to distinguish between the various levels complicates the notation. Fraenkel's solution is to add an axiom (the axiom of replacement) that embeds the concept of logical types, leaving the axiom which defines sets (the axiom of separation) untouched. Zermelo's (and Suppes') solution embeds the concept of logical types into the axiom of separation. Of all the choices, the author prefers Zermelo's for the following reasons. Firstly, it embeds all necessary information without unnecessary additions to the number of axioms or to the notation. Secondly, in the universe of discourse of design information, as will be seen, only a very few levels of logical types are needed, and distinguishing between their elements is relatively easy; it seems unnecessary to include all of the theory of logical types, which is, after all, intended to distinguish between entities that would be difficult to differentiate otherwise. Thirdly, and most importantly from our point of view, the Zermelo solution, flows quite naturally from design considerations and is a natural form of expression

of types of information relevant to design.

8.2.3 Fundamental Structures and Isomorphisms

The fundamental logical structure in HM is an *object*. An object captures a unit of information that is meaningful to a designer. Thus, an object is the formal representation in HM of the informal notion of a design entity. Sets will be used to represent objects. The fundamental isomorphism of HM, then, is that a design entity corresponds to an object (or set). The isomorphism is not part of HM itself (or ZF for that matter), but is an extralogical relationship discerned by the author between design and formal logic, and which gives meaning to the formal system (HM) from the point of view of design.

A bolt, a truss, an airplane, a hole and a run of a finite element program would all be represented by objects. Objects may "contain" other objects (this is discussed below). The entity represented by an object need not be physically realizable (for example, a hole or a fillet), thus it includes *features* [42,53]. Objects may represent machining and other manufacturing processes.

The use of objects is important because it permits *encapsulation* of information, i.e., the discretization of a quantity of information into meaningful structures that can be treated as single units. Encapsulation leads to the construction of *ordered* collections of information. This can greatly simplify manipulation of the information. For example, the alphabet is a structure containing the ordered sequence of written expressions of the phonemes that compose the English language. Similarly, a *screw* is an ordered collection of information that models a device used in the real world as a kind of fastener.

Let the set of all objects be denoted by $\mathbb{O}$, and let X, Y, Z be members of this set (i.e. individual objects).

Axiom 1 (Uniformity of Structure) *All design entities are represented by objects.*

Axiom 2 (Uniqueness of Object Identifiers) *A unique object has a unique identifier.*

Although the relevance of axiom 2 may seem at first glance to be trivial, there is also a more basic, philosophical concern. We must be able to identify any design entity if we are to use it. The process of identification is essential in distinguishing between entities in the universe of discourse. The manifestation of the process of identification is the attachment of an identifier to an entity. Since objects model design entities directly, we *must* also be able to identify objects.

At a more practical level, an object is a conceptual tool that permits us to abstract, infer, and deduce information about design entities, and to classify them by their conceptual definitions.

One of the principal concerns in any theory of information is that of ordering or organizing the information. That is, the definition of relationships between entities is of primary importance. It is a means of making *explicit* information that would be otherwise only *implicit* within a collection of data. In HM, this is done with *relations* and *functions* as defined in ZF.

A relation is a statement that defines a relationship between entities. Given a collection of sets $(A, B, C, \ldots)$, a relation R applied to the collection yields a set of ordered sequences $(a, b, c, \ldots)$ such that $a \in A$, $b \in B$ and so on. The ordered sequence is often used as a representative notation for the relation itself. That is, (x, y) (where $x \in A$ and $y \in B$) represents all ordered pairs arising from the application of some relation R on two sets A and B.

A function is defined the same as a relation, with the added restriction that the relation R can map a single value of $x \in A$ to exactly one member of $y \in B$. Functions are often written $f : X \rightarrow Y$ and are read "f is a function that maps the members of set X to the members of set Y" [80].

It is noted that functions and relations as defined within ZF provide the formal grounds not only for mathematical functions and relations as they are understood outside the field, but also for relations in relational databases, methods in object oriented systems, procedures and routines in conventional programming languages and links in hypertext. They are also essential to the development of data modeling languages such as "Z" [165] and EXPRESS, which is the base language for the PDES/STEP[1] project.

Functions and relations are used to order members of sets, and their formalization is a key part of the author's work. This further extends the isomorphism between set theory and design information. HM currently supports five ordering mechanisms for design information based on functions and relations. They are discussed in Chapter 10.

[1]PDES is the American Product Description Exchange Standard project; STEP (Standard for the Exchange of Product Model Data) is its European equivalent.

8.3 Summary

This Chapter has introduced the fundamentals upon which HM is constructed. An examination of the design process (Section 8.1) indicates that the separate treatment of design information and the processes that act on that information is possible and desirable. Furthermore, an understanding of relevant design information is a necessary prerequisite before an analysis of the design process itself can be attempted. Conceptual notions of logic, particularly than of logical types from set theory, are introduced as relevant building blocks from which HM is developed. The fundamental structures and isomorphisms of HM are introduced. In particular, the notions of a design entity and an object are introduced as the atomic information units from which design models are constructed.

Chapter 9

The Structure of Objects

9.1 Introduction

In this Chapter the nature of individual objects and their internal structure is explored and defined. The treatment is analogous to the basic definitions of sets in ZF. Only a few new terms and special sets are introduced to make the resulting theory specific to engineering design. This too is acceptable from a set theoretic point of view. No new primitives, quantifiers or connectives are introduced, thus maintaining logical validity. The special sets – A, D, O and R – are needed only to distinguish between the various base entities in HM; they will be defined as they are introduced in the text.

9.2 Definition of Objects

A design entity is defined by its observable, or otherwise known, *attributes*. Attributes define the structure of, and function provided by, the entity. For example, a tree is defined by its shape, size, strength of the wood, etc. In fact, the concept "tree" is really nothing but a label attached to a set of observed attributes that are shared by all trees [131, 166]. Attributes are important in design because they model identically the properties of entities in the real world (as opposed to the perceived, conceptual or other worlds).

Let the set of all attributes be denoted by A, and let a, b, c denote members of that set.

Definition 1 (Definition of an Object by its Structure) *An object is a* set *of unique, identifiable, measurable* attributes.

$$\forall(X)\,[(X \in \mathsf{O}) \Rightarrow (\mathrm{SET}(X)) \bullet (\forall(a)\,[(a \in X) \Rightarrow (a \in \mathsf{A})])]\,. \tag{9.1}$$

Since A *is the set of all attributes, we can also write this as:*

$$\forall(X)\,[(X \in \mathsf{O}) \Rightarrow (X \subset \mathsf{A})]\,. \tag{9.2}$$

The set of all attributes A can be regarded as the set of all design variables (like Suh's design parameters [31]). Objects may then be seen as a natural form of grouping and treating these parameters.

A *unique* design entity is one whose attributes differ in some way from the attributes of all other design entities. If two attributes (from two different design entities) are the same, then any operation that can be performed on the attributes will yield the same results. Since a design entity is a set of attributes, an operation applied to a design entity will yield unique results only if there is at least one attribute with a unique value in that entity. Therefore, design entities may be equated by examining the results of the application of operations to them, rather than examining their internal structure directly. Since objects in HM model design entities, we have the following axiom:

Axiom 3 (Identity of Objects) *If the sets of all attributes of any two objects have identical members, and if corresponding attributes in each object have equal values, then the two objects are* identical.

$$\forall(X)\,[\forall(Y)\,[(X = Y) \equiv (\forall(P)(P(X) \equiv P(Y)))]] \tag{9.3}$$

where P is any unary predicate.

We note that axiom 3 is the same as the Axiom of Extensionality in ZF [80], i.e.:

$$(A = B) =_{df} \forall(z)((z \in A) \equiv (z \in B))$$

but is derived from design considerations rather than purely mathematical considerations.

9.3 Views of Objects

A number of properties of objects are very important from the point of view of design; these properties reinforce the isomorphism between set theory and design information. In this Section and those to follow, these properties will be introduced and examined.

The first property is termed *relevance* by the author, and it is manifested as *views* of objects.

One of the most useful innate human intellectual abilities is to mask out certain aspects of an object in favor of other aspects of the object that are of importance to us. For example, a person could very easily sort a collection of books by their size, though, if asked shortly thereafter, be completely unable to describe the colors of the books' covers. Being able to selectively ignore or recognize information lets us isolate and focus our attention only on areas of interest. The importance of filtering information in design environments has been recognized in the literature [17, 31].

A view of an object partitions its attributes, making only some visible and manipulable. Views do not affect the object itself, but establish a projection of the object wherein only certain attributes are accessible. A view partitions the attributes of an object according to criteria explicit in the view itself. After partitioning, the remaining attributes form a subset of the attributes of the object being viewed, that is, a *view* object. An attribute may be active in a number of views but need not be active in all views.

Epistemologically, we can also make the following argument to support this approach.

An "ideal" object is one that models a design entity in every detail, property, behavior, etc. Such a detailed model of reality is unlikely to be possible to construct, yet we can imagine it from a theoretical standpoint. In fact, we can likely not even form such a model mentally, but we can imagine that such models might exist. From a design point of view, not only is it likely impossible to construct such ideal models, but it is also unwarranted. In design, we are specifically concerned with only subsets of all the attributes of a design entity. Thus, the logical notion of a view permits us to project an ideal model of a design entity onto a relevant design model of that entity.

We see, then, that the isomorphism between set theory and design forms a connection that extends from

the very notion of existence to a specifiable formal model of existence.

If we consider a particular object to be a "complete" model of some design entity, then any proper subset of the members of the object can be considered a specific view of the object with respect to the selected members.

Using the definition of subsets in [80], we can write in the notation of HM:

$$\forall(X)(\forall(Y)\,[(X \subset Y) =_{df} \forall(x)\,[(x \in X) \Rightarrow (x \in Y)]]).$$

If $(X \subset Y)$ but $(X \neq Y)$, the subset is a *proper subset*. The number of possible views of an object is the cardinality of the power set $\mathcal{P}(X)$ of the object[1].

Many of these views would be trivially unimportant to a designer. But there is no way to define *a-priori* only the views that are relevant. Yet, we can restrict our definition of a view in a manner similar to the way that the definition of a subset is restricted in ZF. This kind of restriction also happily prevents certain kinds of logical paradoxes that would render the theory as a whole invalid.

We begin with what is generally called the Axiom of Separation.

$$\exists(S)\,[\forall(x)((x \in S) \equiv ((x \in A) \bullet \varphi(x)))]$$

where there are no free occurrences of S in φ. This says that for *any* set A, and any propositional function (i.e. predicate) φ, there is a set S that is a subset of A and that contains only members of A that satisfy φ.

When we say that there can be no free occurrences of S in φ, we mean only that φ must not contain occurrences of $\exists(S)$ or $\forall(S)$ since this would imply that S is defined in terms of itself and would lead to paradoxes. This is not a real problem in HM itself, because it would be meaningless to define a view with respect to itself, so a designer would likely never attempt it. However, it is enforced in HM for completeness and consistency.

We refer to the Axiom of Separation as an *axiom-schema* because the symbol φ represents a *group* of

[1] The power set is a well-defined entity in classical set theory [80]

predicates. We may write the set of all predicates φ as Φ. The equation above therefore actually represents a group of axioms, each having a different predicate substituted for $\varphi \in \Phi$. In Section 8.2.1.2, it was explained that the restricted nature of the universe of discourse of HM lets us investigate the nature of entities within that universe much more closely than is possible in ZF. Here is one example of the degree of detail that is possible: in ZF, little can be said about the actual predicates that can be substituted for φ in the Axiom of Separation; but, as we shall see below, in HM we can investigate a number of important groups of predicates that apply to design. Views are the first such case.

We interpret the Axiom of Separation for HM views as follows: For an object X and any predicate φ, there is another object Y whose set of attributes is a subset of the attributes of X, all the members of which satisfy φ.

The Axiom of Separation is an axiom schema; in HM the axiom of views is in fact a subset of that of the Axiom of Separation. It can be written as follows:

Axiom 4 (Axiom (Schema) of Views)

$$\exists(Y)\,[\forall(a)((a \in Y) \equiv [(a \in X) \bullet \gamma(a)])] \tag{9.4}$$

where γ contains no free occurrences of Y.

γ is a new symbol, and is used to represent a *subset* of all possible predicates φ that satisfy the Axiom of Separation. In particular, γ represents a predicate that "defines" a view; different γ predicates will produce different views. γ, then, is the criterion by which a specific view is defined. These criteria are attribute-specific. For example, if γ were such that only attributes that modeled spatial dimensions satisfied it, the resulting view of an object would be its 3D geometric representation.

Let the set of all views be denoted by Γ, and let γ be a member of that set.

The notion of a view being a subset of an object is captured by the following definition.

Definition 2 (Views) VIEW( ) *is a binary function whose parameters are an object and a view criterion specification, and whose result is another object called a* view object *whose attributes are a subset of the attributes of the input object selected according to the given criterion.*

$$\forall(X)\,[\forall(\gamma)(\exists(Y)\,[(Y = \mathrm{VIEW}(X,\gamma)) \bullet (Y \subset X)])]\,. \tag{9.5}$$

We can write this in a functional notation as

$$\mathrm{VIEW} : O \rightarrow \mathcal{P}(O).$$

Objects that are the same in every regard are identical. This applies to view objects as well. However, this notion also suggests a relationship between the objects from which the views were generated. If there exists a view object that can be derived from two non-identical objects through the use of a single view criterion γ, then we define the two non-identical objects as being *similar objects*.

The inclusion of similar objects in HM is motivated by the observation that for many design tasks, only a certain view of an object is sufficient to permit completion of the task. Since views are projections of objects, it becomes important to be able to make statements about the objects that give rise to such view objects.

Theorem 1 (Similarity of Objects) *Two objects are* similar *if the application of a given view criterion γ to the objects produces identical view objects.*

$$\forall(X)\,[\forall(\gamma)\,[\forall(Y)](X \sim Y) \equiv (\mathrm{VIEW}(X,\gamma) = \mathrm{VIEW}(Y,\gamma))]]\,. \tag{9.6}$$

The symbol $\sim$ is used to denote similarity of objects.

The author's motivation to have views of design information is five-fold. First, we have the epistemological argument presented earlier. Second, completeness requires that HM extend to cover the entire universe of discourse; and in a design environment, the universe of discourse includes views as relevant design entities. Third, from the standpoint of conciseness, views permit a structure to exist in the simplest form that maintains its semantics. Fourth, from an organizational standpoint, views permit information to be

ordered by its relevance to a task. Fifth, and lastly, views implement information hiding, which is desirable practically for a number of reasons. The designer will have a simpler task if only relevant information is visible. Information selected by view can be presented to the user in a more understandable form. Superfluous information can be excluded to increase efficiency and robustness. More practically, views can provide uniform interfaces to parts of a database even though the internal structure of the objects in the database may change.

Views are an especially powerful tool when defined as components of attributes. For example, an attribute occurring in two different views may indicate coupling between the views. Alternatively, the sets of views of two objects can be intersected, whereupon the cardinality of the intersection set can be used to measure functional or other coupling. Similarly, small sets of views can be used in order to study how different types of attributes affect the coupling of two objects.

9.4 Domains and Ranges of Attributes

9.4.1 Set Theoretic Foundations

The second important property of objects has to do with the structure of the attributes that compose them. Here, we introduce the necessary set theoretic background to formalize object attributes in HM. We begin by considering the formal definition of a *relation on sets*.

$$\forall(x)\,[(x \in R) \Rightarrow (\exists(u)\,[\exists(v)(x = (u, v))])]$$

where x is an ordered pair, $u \in U$ and $v \in V$ (U and V are sets) and R is a relation.

This is the definition of the *cartesian product* $U \times V = R$. The domain and range of R are given by:

$$\begin{aligned} \mathrm{dom}(R) &=_{df} \{x : \exists(y)(\langle x, y\rangle \in R)\} \\ \mathrm{ran}(R) &=_{df} \{y : \exists(x)(\langle x, y\rangle \in R)\}\,. \end{aligned}$$

Let D be the set of all possible attribute domains and R be the set of all possible attribute ranges. We may now write:

$$\mathsf{A} = \mathsf{D} \times \mathsf{R} \tag{9.7}$$

and $\mathrm{dom}(\mathsf{A}) = \mathsf{D}$ and $\mathrm{ran}(\mathsf{A}) = \mathsf{R}$. Also, because of definition 1, we can write for an object X:

$$\mathrm{dom}(X) \subset \mathsf{D} \tag{9.8}$$

$$\mathrm{ran}(X) \subset \mathsf{R}. \tag{9.9}$$

9.4.2 Domains and Ranges in HM

An attribute representing a property or behavior of a design entity is specified by two pieces of information. First, the concept that typifies the attribute is needed: its *domain*. In the most general sense, domains of attributes can include integers, real numbers, text, arrays, etc. The domains of attributes relevant to engineering design are discussed below. Second, a specification of how the property is exhibited by a particular entity is required. The set of possible values that an attribute can have is called the *range* of the attribute.

Definition 3 (Domain of Attributes) *The* domain *of an attribute is the abstracted, observable, quantifiable property of a design entity that the attribute represents. The domain of an attribute includes an associated dimensional unit.*

Definition 4 (Range of Attributes) *The* range *of an attribute is the set of all values that are meaningful within the domain of the attribute, and any one of which may be the actual value within an arbitrary object containing that attribute. The set of values can be discrete or continuous, single-valued or multiple-valued.*

The dimensional units mentioned in these definitions are discussed below.

Let D be restricted to the set of all attribute domains in HM only, and let R be the set of all attribute ranges in HM only.

Definition 5 (Attributes) *Attributes are ordered pairs* (d, r) *where* $d \in \mathsf{D}$ *and* $r \in \mathsf{R}$*, and the set of all attributes is the cartesian product* $\mathsf{D} \times \mathsf{R}$.

The definition of the cartesian product and its relationship to ordered pairs is defined within ZF. We use ZF here to provide a formal definition of attributes.

Theorem 2 (Identity of Attributes) *Two attributes are identical if their domains are identical and their ranges are equal.*

9.5 Dimensions of Measurement

There is an important observation that must be made at this point regarding attributes for engineering design. To be meaningful, attributes must not only be observable; they must also be measurable. If an attribute is not measurable, its value cannot be compared to other values or used in computation, and would hence be comparatively meaningless. Therefore, the domains of attributes must include *dimensions of measurement* against which the attribute can be compared. This is another important property of objects in HM.

In order to satisfy the condition of measurability of attributes given above, the author has defined the members of the set of attribute domains D to contain dimensional properties. The members of D in HM are: length, mass, time, cost, quantity (or enumeration), NDU (non-dimensional units, for ratios, etc.) or any combination of these (e.g. velocity, energy, and so on). Although only length, mass and time are commonly considered, the author has elected to add other dimensions because of their relative importance in engineering environments. This approach is far more powerful than schemes that only represent numeric quantities because it is a natural form of expression that is physically meaningful, and because it captures all the necessary semantics of dimensional standards at the axiomatic level. For example, correct dimensional analysis becomes an inherent property of HM. Dimensional information has also been found to be of great assistance in dealing with spatial constraints [167].

The members of R, the set of ranges, in HM are: integers ($\mathcal{I}$), real numbers ($\mathcal{R}$), boolean values ($\mathcal{B}$) and text ($\mathcal{T}$). The author is undecided as to whether complex numbers should also be included as possible

range values: though they are of use in many design methodologies (particularly in the area of analysis), they can also be considered as composites made up of two real numbers.

The author believes this approach to be superior because it is a very natural form of expression that is physically meaningful, and it captures all the necessary semantics of dimensional standards at the axiomatic level.

9.6 Constraints and Attributes

Constraints are the principal driving force of the engineering design process. They are manifested as relationships between information regarding design entities. Insofar as constraints are vital to design information specification, they must be represented by HM.

As with views (Section 9.3), constraints operate at the attribute level within HM. Both attribute domains and ranges may be constrained. Attribute domains may be constrained to be of specific types (e.g. modular assembly components might have a constrained kind of shape or material), and attribute values may be constrained to be constant-valued, single-valued or multiple-valued, continuous or discontinuous, and so on.

The issue of constraints in design is far more complex than may be implied here. In general, the constraint satisfaction problem is characterized as NP-complete [55–57], which means the time required to solve the problem varies exponentially with the size of the problem. For even small design problems, the required computation can be intractable. However, the *process* of constraint satisfaction is a component of the design process itself, and therefore falls outside the bounds of the immediate concern of the author in this work. While this simplifies our task, we recognize that more work is needed before HM can support constraints appropriately. However, it should be clear that the *specification* of constraints, in the form of functions and relations that define subsets of attributes and objects, is inherent to HM.

It is noted that the set $\mathbb{D} \times \mathbb{R}$ (discussed above) contains attributes that are meaningless in a design environment. For example, an attribute with the domain of *quantity* cannot have a range within the set of real numbers. Clearly, some constraints will be necessary just to keep a model consistent with respect to attribute definition.

9.7 Summary

In this Chapter, the basic structure of individual objects and their internal structure has been presented formally. An object models a design entity, and is defined as a set the members of which are attributes. Each attribute has a domain, a range, and an associated dimension of measurement. A view of an object is a subset composed of attributes of the object that satisfy some predicate. Views permit the isolation of relevant attributes based on externally supplied criteria. Various primitive relationships between objects (e.g. identity, similarity) are also formailized. Three which issues remain for which further research is indicated. First, the role of views has been identified as key to a number of application domains, especially that of database design for engineering environments [94, 168, 169]; a detailed study of views could be highly beneficial to such efforts. Second, the unique approach taken with regards to dimensions of measurement should be investigated more fully. Third, the pivotal role played by constraints in the design endeavor makes their further study relevant and important.

Chapter 10

Ordering Mechanisms for Design Information

10.1 Introduction

In this Chapter, the author identifies various ordering schemes (*abstraction mechanisms*) that can be imposed on engineering information to make semantic content explicit. These mechanisms are derived from considerations of the types of information available to a designer and are defined within a set theoretic framework (i.e. HM). They represent primitive ontological notions based on how designers regard the universe. These notions cover the classification of design entities according to various criteria; though extralogical in and of themselves, these notions are well founded in empirical research in philosophy, psychology, and artificial intelligence [28, 131, 166].

The matter dealt with in this and the following Sections occupies a higher level of abstraction than the matter of Chapter 9: we discuss statements about *collections* of objects, rather than just individual objects. These collections are (classical) sets of objects, and they obey all the notions and axioms of ZF set theory.

Objects are organized by establishing relationships between them. The kinds of relationships are independent of the objects that take part. Consider, for example, $X = f(Y)$, where X and Y are objects. The function f may be applied to many objects, and yield many objects. It thus defines sets of objects C_x and

C_j such that $X \in C_i$ and $Y \in C_j$. The organization of objects, then, proceeds by defining the relations that in turn define sets of objects.

We recognize now that the set Ω, the set of all objects, is one of these collections. The relationship between the members of this set is that every member is an object. Similar arguments may be made for each of the sets introduced in Chapter 9.

10.2 Types of Objects

One of the most natural and useful abstraction mechanisms for ordering (or classifying) entities is by structural (as opposed to behavioral) similarities of the entities. Sometimes called "classification" or "typing", this mechanism is also a basis of human cognitive function in general. Ordering schemes of this type are only partial because many such schemes may be imposed on the same collection of entities, each yielding a differently ordered collection. The human mind thinks about entities so grouped by thinking about an abstracted (or *generalized*) entity that captures only that which is common to the members of the collection and leaving indeterminate (or at least variable) other aspects of the members of the collection. In HM, the generalized, conceptual entity meant to represent a partially ordered collection of design entities is called a *type*. The partial orders that types impose on collections of entities define relationships shared by the members of the collections, thus making information regarding the members explicit. Since objects in HM model real world entities, and since objects are defined in terms of attributes, types in HM must model relationships between objects by modeling relationships between attributes of objects.

The distinction between a type and the collection of objects that the type models must be kept clear. A collection of partially ordered objects is, essentially, a set of sets, and hence exists at a different degree of abstraction (or logical level) than do objects. A type, on the other hand, *models* a collection of objects, and therefore exists at the *same* logical level as do objects. This distinction becomes crucial if we are to insure that HM is consistent. The notion of logical levels descends from Russell's Theory of Logical Types [153] and is a generalized mechanism to distinguish between sets, based on the degree of abstraction required to create the sets. The Axiom-Schema of Separation, as it is used in [1, 80, 153] and by the author, implies this same distinction between logical levels. Thus, ZF supports the distinction of logical levels according to Russell's theory. The distinction is also important to HM.

The key to providing a *consistent* theory of design information lies with the Axiom of Separation, which we have already seen in Section 9.3. Again, our approach is to interpret the Axiom of Separation in terms of design considerations and eventually draw forth a group of predicates that can be substituted for φ in the Axiom which has meaning from a design standpoint.

For a given object, we might expect predicates such as *"This entity has a threaded shaft"* or *"This entity is made of cold-rolled steel"*. For a type collection, we might expect predicates such as *"This entity is a bolt"* or *"This entity occupies space and is made of metal"*. In considering type collections, we disregard any predicates that depend on the values of attributes of objects; i.e. we know at once that a predicate such as *"This entity is 3.5 centimeters long plus or minus 1 millimeter"* applies to an object, while a predicate such as *"This entity has a dimension that we call its length"* is clearly a predicate on a type. The distinction is that the former is a predicate on a design entity and the latter is a predicate on an abstract entity that generalizes some aspect of the former. Also, the former mentions an attribute domain (length) and a range value (3.5 centimeters ...) while the latter mentions only a domain.

In other words, the process of generalization of attributes involves neglecting the values of the attributes and dealing only with their domains. That is, attributes that are predicable on type collections are not predicable on design entities or the objects that model them, but on *generalizations* of the design entities.

In a more practical sense, we may state this as follows:

In HM, objects are *typed* (or classified) by their structure: similarities in structure are expressed by similarities in the domains of the attributes of objects. Attributes of objects are quantitative measures and represent in HM only those quantitative aspects of design entities.

The criterion used to define a collection of objects is based on attribute domains. An object is included in the collection if the domains of all its attributes map identically to all the domains in the criterion. Information defining the criterion is supplied by the type.

Axiom 5 (Abstraction of Structure) *Abstraction of object structure is based on generalization of object attributes and results in types, which are objects that model collections of objects that share structural features.*

Let a type collection be denoted by C_T.

Definition 6 (Type Collections) *A type collection is a set of objects, the domains of the attributes of all the members of which are the same.*

$$\exists(C_T)\,[\forall(X)\,[\forall(Y)\,[((X \in C_T) \bullet (Y \in C_T)) \equiv (\mathrm{dom}(X) = \mathrm{dom}(Y))]]]\,. \tag{10.1}$$

We note that the phrases $(X \in C_T)$ and $(Y \in C_T)$ obey the rules of logical types (see Section 8.2.2).

We can represent all the information necessary to define a collection C_T for types by means of an object. Specifically, for a type collection C_T, a type T is defined as an object whose domain is the same as the domain of every member of C_T. This defines the minimum necessary information to capture the notion of type collections.

Because types and type collections are based on object domains only (i.e. range information is ignored), then *any* member of a type collection C_T can fill the role of type T. That is, for a type collection C_T, we may take any member object from C_T and successfully use it as a type for that type collection. Thus type objects T are not actual objects distinct from other objects, but are rather ordinary objects that we consider in the *role* of representatives of their type.

We may denote the set of all types by T, therefore $T \in \mathsf{T}$ and $\mathsf{T} \subset \mathsf{O}$.

We define the binary predicate IS_A to capture the type relationship between two objects.

Definition 7 (The Typing Predicate)

$$\forall(X)\,[\forall(Y)\,[\mathrm{IS\_A}(X,Y) \equiv (\mathrm{dom}(X) = \mathrm{dom}(Y))]]\,. \tag{10.2}$$

The author notes that if Y were defined to be the type for its type collection, then IS_A could be used to determine if arbitrary objects were "of a given type".

Furthermore, it is redundant to have more than one type object for a given type collection. However, nothing has been said yet that would prevent two objects of a given type collection from being considered types. We can capture the uniqueness of type objects using a notation suggested in [1] to indicate the existence of unique individuals.

Theorem 3 (Theorem of Type Uniqueness)

$$\forall(C_T)\,[!\exists(T)(T \in C_T)]\,. \tag{10.3}$$

The notation $!\exists$ means "*. . . there exists exactly one . . .*".

Now let us return to the Axiom of Separation. It is written:

$$\exists(S)\,[\forall(x)((x \in S) \equiv ((x \in A) \bullet \varphi(x)))]$$

and states that there exists a subset S of a set A all the members of which satisfy φ (with no free occurrences of S in φ).

With regard to types, the author interprets the axiom as follows: there exists a subset X of the set of all objects O all the members of which satisfy a predicate θ. In this case, θ is the predicate that differentiates objects by type – in other words, IS_A. We can then write the Axiom of Separation for types in HM as the following axiom schema:

Axiom 6 (Axiom of Types)

$$\forall(Y) \qquad \exists(C_T)\,[\forall(X)((X \in C_T) \equiv [(X \in \mathsf{O}) \bullet \mathrm{IS\_A}(X, Y)])]\,. \tag{10.4}$$

The space separating "$\forall(Y)$" from the rest of Equation 10.4 is a convention used in symbolic logic to indicate the extent of an axiom with respect to a set of entities. In this case, it binds the use of Y in the axiom so as to explicitly define the range of values that Y can attain within the axiom.

Types themselves represent our abstract concepts of design entities based on their quantifiable attributes; they define the properties of a set of design entities without defining the degree to which each real world entity exhibits those properties.

In attempting to relate objects and types, the distinction between types and the collections of objects that types model is essential. Collections of objects are not directly comparable to objects because they are of different degrees of abstraction (see Section 8.2.2); that is, apparently intuitive statements such as $X \cap C_T$

and $X \in T$ are invalid, the system resulting from their inclusion would become inconsistent, and there would be no way of assuring that all statements that can be formulated in HM can be proved. On the other hand, statements of the form $\mathsf{O} \cap \mathsf{T} = \mathsf{T}$, or $X \in C_T$, are acceptable because O and T, and X and C_T, are of the same degree of abstraction.

This is a good example of the power of set theory: it forces us to think more clearly by giving us a system wherein logical errors are more easily detectable, without restricting our freedom to express consistent, relevant information. The majority of logical errors are detectable because "statements" in HM are theorems; if they can be proved within the system, they are valid (correct). Not all errors can be caught this way because of the inherent incompleteness of formal systems (see Section 5.1).

The relationship between a type and objects of that type (the *type-object* relationship) is a one-to-many relationship. But the relationship between an object and its type (the *object-type* relationship) is one-to-one. We would have to be outside the system to "see" that the current object's type has more than one instance; this would introduce self-reference that we want to avoid. Because one-to-one relationships are dealt with in a straight-forward fashion with functions, and because of the added complexity of dealing with both objects and type collections of objects, HM models objects as having types, rather than types as having collections of objects.

Furthermore, types collections (C_T) are not considered to be primary entities in HM. Design entities are the only entities of primary importance, since they are identified with real-world entities relevant to designers. Types collections, though essential on the theoretical grounds discussed above, are *derived* from objects. They are thus regarded as mutable, built from objects that are known or presumed to exist in some way. This must be the case because the underlying requirements of the criteria for the formation of types can change during the course of a design process, or as our collective understanding of design evolves. If we were to define types as immutable structures, we would be locking ourselves in to a particular viewpoint of the nature of design which might turn out to be insufficient.

10.3 Aggregations of Objects

We have seen that types permit the ordering of objects, using abstraction of the attributes of objects.

An aggregate is also an ordering device for objects, but it works through a different abstraction mechanism: recursive containment. Put simply, an aggregate is just a collection of objects. Membership in aggregate collections is based on entirely arbitrary criteria that depend on the exigencies of the problem at hand. This distinguishes aggregates from types, for which the defining criterion is precise and known *a-priori*.

Aggregates permit the hierarchical ordering of objects of the same degree of abstraction (i.e. aggregates cannot mix objects and types). Design is strongly hierarchical. Parts may be assemblies of other parts. Even processes (for example, a finite element analysis, or a machining process) are composed of subprocesses. The larger the design problem, the more important hierarchical ordering becomes.

Once again, we rely on the Axiom of Separation to guide us in formalizing our notion of aggregates. In this case, we write the Axiom as follows:

Axiom 7 (Trial Axiom-Schema of Aggregates) *There exists a collection of objects C_A, all the members of which satisfy a particular predicate δ,*

$$\exists(C_A)\,[\forall(X)((X \in C_A) \equiv [(X \in \mathsf{O}) \bullet \delta(X)])]\,. \tag{10.5}$$

C_A is an aggregate collection, and δ is one of a set of predicates Δ used to define membership in the aggregate object that model these collections.

However, we observe that (1) C_A and O are *collections* of objects; and (2) δ applies only to single objects in C_A. Defining δ in terms of C_A breaks the restriction of free variables on the axiom of separation. Therefore, axiom 7 itself is not enough to define the relationship between members in an aggregate.

The generally accepted theory of *classes* of sets [170], which accounts not only for sets but also for collections of sets, achieves nothing for us except the replacement of predicates like δ by classes of objects that are defined in terms of δ.

Alternatively, we might consider defining δ in terms of $\bigcap_i c_i(a_i)$ where $a_i \in C_A$ and c_i is some predicate constraining a_i. But in this case each constraint depends on only one member of C_A, whereas we need a single relationship over all the members of C_A.

To solve this problem, let us begin by saying that a relationship is needed to define the nature of the components' use in the assembly. The relationship takes the form of a constraint on the attributes of the

assembly (an aggregate object). Thus, the constraint is at the same logical level as the aggregate, *not* at the level of the objects that compose the aggregate. Let κ represent these constraints.

In general, a constraint of this kind on an aggregate will not act on all the attributes in the aggregate. For example, a constraint defining the relationship between two links in a four-bar linkage need not act on the other links. So, in general, a constraint will act on a subset of the attributes of an aggregate.

Now, a view (see Section 9.3) is a subset of an object. If we define a certain view of the aggregate to contain only the attributes acted upon by some aggregation constraint κ, we may then say that the constraint acts on *all* the attributes *of the view*. Now, we can begin writing another axiom, based on views of an aggregate. Each constraint κ can be used to define a view criterion γ, which in turn defines a view. Thus a collection of constraints κ_i for an aggregate leads to a collection of view criteria γ_i and a collection of view objects V_i. We now define C_A more precisely: $C_A =_{df} V_i$.

Axiom 8 (Axiom-Schema of Aggregates) *There exists a collection of views C_A of an aggregate object, all the members of which satisfy a particular predicate δ.*

$$\exists(C_A)\,[\forall(V)((V \in C_A) \equiv [(V \in \mathbf{O}) \bullet \delta(V)])] \tag{10.6}$$

where $\delta(V)$ must be true for all views in C_A; that is,

$$\delta(V) =_{df} \bigcup_i \kappa_i. \tag{10.7}$$

This equation will be true if at least one κ_i is true. It is noted that this is counter-intuitive: we might have assumed that all constraints must be simultaneously true (i.e. $\bigcap_i \kappa_i$), but this is not the case.

Since attributes appearing in more than one view object of an aggregate are identical (not only equal or equivalent – see Section 9.3), we can create the aggregate object itself by simple union: $Y = \bigcup C_A = \bigcup_i V_i$.

A graphical depiction of the aggregation of four links to compose a four-bar linkage is given in Figure 10.1, including the link objects `ground`, `input`, `coupler` and `output`, all views, view criteria γ_i and constraints κ_i.

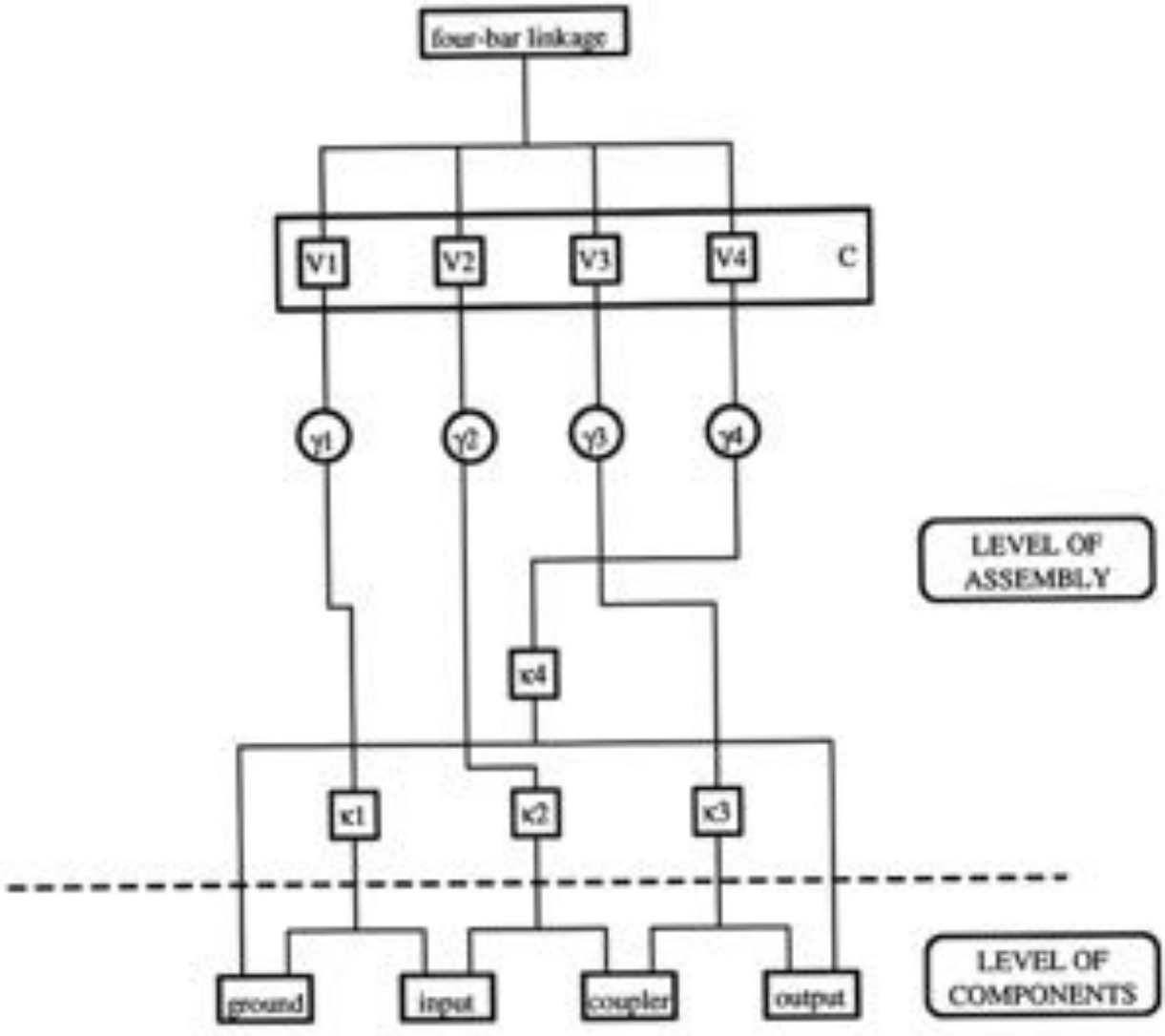


Figure 10.1: Aggregation of a Four-Bar Linkage in HM.

The predicate set Δ is an important one: its elements provide the means of defining all the necessary relationships in an aggregate object. The consequences of this statement are made quite clear by considering a simple physical assembly of components. Objects would model each component. An aggregate object would model the assembly. It would be the role of the δ predicate for that assembly not only to partition objects according to which are needed in the assembly and which are not, but also to provide the exact relationship that exists between the components in the assembly. In other words, it is the aggregate predicates Δ that permit the definition of how components in a physical assembly mate, the tolerances of the fit, the manner in which the mating occurs (the assembly process itself) and so forth.

An aggregate object need not contain *only* other objects, but can itself have attributes. One obvious example of the kind of attributes that an aggregate itself might have are size and shape. These clearly cannot be derived from the components of the aggregate alone (e.g. information about a bolt tells us nothing about the assembly in which the bolt is to be used). This too is permitted by HM. Aggregate objects, then, are not just sets of objects, but an object with attributes whose values are other objects.

At this juncture, we could begin a detailed study of the properties of the predicates in Δ, but shall not. This document is meant to be both an overview of HM and a statement of the *fundamental* axioms that compose it. In this spirit, then, we defer such discussions to future work.

10.4 Classes of Objects

In addition to classification by structure using types, objects can be also classified by the function they are meant to provide. The importance of capturing semantic content of function is best exemplified by conceptual design.

Conceptual design is one of the first steps in a design process, and has the greatest impact on downstream decisions [17,42,171]. In general, conceptual design is considered to be the mapping between the function provided by some entity and the physical specification of the entity. Very little is known about conceptual design and we do not presume a simple solution to the problem here. However, the author has devised a mechanism to ease the development of a system of classification by function.

The mapping between structure and function is not necessarily one-to-one: a particular structural com-

ponent may provide more than one function, or vice versa. The mechanism of types discussed in Section 10.2 is inappropriate: the mapping between objects and types is one-to-one. Therefore, the structural properties of the entity do not capture semantics of function. A mechanism different from typing is required.

The author defines a *class* of objects as an aggregate object whose members all represent design entities that exhibit a given function. HM considers inclusion of an object in a class sufficient to establish that the object exhibits a given function.

It is unclear to the author at this time whether a class should include information for modeling of the function itself, or whether this information should be contained by the member objects, or by the δ function of the aggregate that relates them. The relationship modeled by the function would permit access to the members of the class, much as IS_A provides, but without the constraints that IS_A imposes on the attributes of member objects.

10.5 Specialization and Generalization of Objects

10.5.1 Specialization of Types

The abstraction mechanism of *specialization* is implemented in HM by *inheritance of* type. Inheritance is a mechanism similar to aggregation, but is controlled by the operation of set *union*.

The difference between inheritance and aggregation is very important from a semantic point of view. For example, to say that an automobile inherits the attributes of its engine (e.g. power) is meaningless; the reason why humans can make sense of such statements is because we can *interpret* it correctly and extract the necessary *implicit* information from the statement. However, this highly informal and subjective approach is very undesirable. The correct statement that can be supported by a formal theory would be that the automobile is an aggregate, one component of which is an engine that has a certain power rating. Because HM is meant to formalize design information, the distinction between aggregation and inheritance becomes essential.

A type, then, is the union of all the attributes of the types that are inherited by it. Union of sets is very

well understood in set theory and provides a simple and rigorous way to formalize specialization through inheritance.

Using the Axiom-Schema of Separation again, we can write the following.

Axiom 9 (Axiom of Specialization) *For a given type T, if there exists a collection of types C_S such that T contains all the attributes of all the members of C_S, then T is said to* inherit *from the members of C_S, and T is a type* specialized *from the types in C_S.*

$$\forall(T) \qquad \exists(C_S)\,[\forall(U)((U \in C_S) \equiv [(U \in \mathbf{T}) \bullet (U \subset T)])]\,. \tag{10.8}$$

We can define the predicate INHERITS as:

Definition 8 (The Inheritance Predicate)

$$\forall(T)\,[\forall(U)\,[\text{INHERITS}(T,U) \equiv (U \subset T)]]\,. \tag{10.9}$$

The phrase INHERITS(T,U) is read "*type T inherits the attributes of type U*".

As in the standard definition of the union operation, duplicate elements are excluded from the union set. The identity axiom (axiom 3) defines the criterion by which duplicates are identified in HM.

In design, specialization is an important mechanism because it permits the creation of specialized types from a collection of more general, abstract types. It is, therefore, a *top-down* procedure.

Design also tends to be a top-down process [43, 49, 69, 106, 126, 167, 172], moving from the general (conceptual design) to the specific (detailed design). This impacts on how we treat design information. Because design begins from the general and moves to the specific, we can expect that at an arbitrary point along the development of a design artifact, information regarding the artifact will be incomplete in detail.

Specialization, then, being a top-down process, is used in HM to permit incomplete information about design entities to be captured in a consistent manner, and to permit the generation of (application) specific types from general types.

10.5.2 Generalization of Types

Generalization is the inverse of specialization, and from the purely theoretic point of view of HM, the relationship between the two is bidirectional (i.e. specialization and generalization are opposites but equivalents). Thus:

Axiom 10 (Axiom of Generalization) *If all the members of a collection of types C_G have some attributes in common, then there is a generalized type T that contains those common attributes, and all the types in C_G inherit the common attributes from T.*

$$\exists(T) \qquad \exists(C_G)\,[\forall(U)((U \in C_G) \equiv [(U \in \mathsf{T}) \bullet (T \subset U)])]\,. \tag{10.10}$$

Generalization is important to design too, for two reasons. The first is quite practical: in an environment where a number of types of design objects have been generated independent of each other, generalization provides a formal mechanism by which we can *unify* our models of the objects. This kind of unification minimizes the amount of information needed to completely specify a design artifact model, making the model simpler and clearer[1].

The second reason that generalization is important has to do with theoretic, taxonomic concerns. One obviously desirable goal in design theory is the generation of usable, globally applicable taxonomies of design entities. The issue of taxonomy in design theory was discussed in Section 4.2. Taxonomies themselves can help standardize our models of design entities and control the information required for the models. Generalization in HM gives us a very specific formal methodology for generating design entity taxonomies. Taxonomies resulting from the application of generalization to types in HM would result in inheritance networks of types that would permit the classification (at least in theory) of arbitrary kinds of design entities. The issues involved in generating such taxonomies are interesting and many, and are not dealt with specifically in this document.

[1] This corroborates Suh's second axiom of information minimization [31].

10.5.3 Relationship Between Specialization and Views

Assume two types, T and U, and a view criterion γ_d that only examines object domains. If the types are similar under the view criterion, then the view objects formed by application of the criterion to the type objects are identical. Therefore, the view object itself is seen as a type from which types T and U inherit attributes. This may be written in the notation of HM as

Theorem 4 (Similarity of Types)

$$\forall(T)\,[\forall(U)\,[\forall(\gamma_d) \quad [(\text{VIEW}(T,\gamma_d) = \text{VIEW}(U,\gamma_d) = V) \equiv (\text{INHERITS}(T,V) \bullet \text{INHERITS}(U,V))]]]\,. \tag{10.11}$$

This further reinforces the importance of views as an organizational mechanism: we see that views can be used to define the conditions whereby inheritance between object types occurs.

10.6 Summary

The author has presented in this Chapter five ordering mechanisms for objects: typing (by structure), aggregation, classification (by function), specialization, and generalization. These five mechanisms provide the means of organizing collections of objects in meaningful ways (in a design context); such organization of information is essential to maximizing the amount of explicit information a designer has available to him/her, which in turn can decrease the chances of *mis*interpreting design information. The ordering mechanisms are based on ontological considerations of real-world entities, not on the empirical evidence as provided by the conventional understanding of design. In this way, independence from design is maintained; we may then be more confident of the universal applicability of these mechanisms.

Chapter 11

Discussion

11.1 General Summary

HM is an evolving, growing structure, but the core of the model as presented here is accurate and will not change as the model develops. HM is a variant of classical ZF set theory, extended and interpreted to suit engineering design information. HM provides the isomorphisms that permit us to view design information in an objective, formal way. Specifically, these isomorphisms permit design entities to be represented by *sets* that can be grouped into subsets using criteria all finding their roots in the Axiom of Separation as defined in ZF; that is, the organizational axioms of HM (Chapter 10) are all axiom sub-schema of the Axiom of Separation.

Other researchers [162, 173] have suggested extending formal systems to suit design, but the author is not aware of any attempts as detailed or well-grounded in accepted logic systems as is HM.

HM is not intended to automate the design process, but rather to provide a structured notation that makes information about design entities clearer and thus permits the designer to apply whatever thought processes he/she prefers. The author perceives the designer as existing in a symbiotic relationship with design tools such as HM, rather than being replaced by them.

HM is based on a functional model of the engineering design process that views design information as separate from the various processes that act on this information during the course of design development.

The key issue that permits HM to be stated at all is that therein the universe of discourse of ZF is restricted to only those entities pertinent to design. In this way, we can make more specific statements (i.e. axioms) than ZF alone permits.

The axioms of ZF are not altered by HM, and no new primitives, connectives and quantifiers are introduced. For these reasons, we are assured that HM represents a valid, consistent formal system of logic [80] (with respect to set theory), specifically geared to design information.

Objects are sets of attributes, and capture meaningful information about design entities. This provides a natural form of expression for design information because objects are conceptually equivalent to the entities they model. Attributes are defined in terms of domains and ranges; domains of attributes in HM include generalized dimensions of measurement. Views permit objects to be partitioned according to criteria specific to particular design tasks. Relations and functions are used to define general classes of operators on objects, and provide a flexible, extensible mechanism for the logical representation of various kinds of relationships and constraints. The specific relationships of structural similarity, functional similarity, aggregation, specialization, and generalization are all captured formally by HM.

Since HM applies to different design tasks, such as solid modeling, analysis, and so on, it presents an *integrated* approach to the specification of design information that is extensible: new entities and relationships can be specified using the model without altering the model itself.

HM provides (a) a basis for building taxonomies of design entities, (b) a generalized approach for making statements about design entities independent of how the entities were generated (i.e. independent of the design process used to create them) and (c) a formal syntactic notation for the standardization of design entity specification.

11.2 Future Work

There are various fronts on which work on HM can continue. The role of constraints in the hybrid model must be examined, and suitable theory generated. A review of the current literature indicates that relatively little work has been done in this area. This may be because the design entities that are constrained have been vaguely and/or imprecisely defined in the past. The formal understanding of these entities provided

by HM may also help us understand the constraint satisfaction problem better.

Classification of design entities by function is another area where current understanding can be improved; emphasis will be placed on this in the future work of the author.

Due to the hierarchical nature of design entities (particularly mechanical design entities), aggregation is a key abstraction mechanism. Further study of the Δ predicates that HM supports for the definition of aggregates is warranted.

At the theoretical level, a formal theory of design information will be a useful tool for the study of the design process itself by providing a uniform lexicon and grammar for information specification. The author notes that it is possible to blend ZF and first order predicate calculus [80]. The predicate calculus is the formal basis of such tools as expert and knowledge processing systems. Also, fuzzy logic [147,148,174] presents a unique opportunity for representing the notion of uncertainty in purely formal terms. It is interesting to speculate on the nature of the combination of these tools with HM, and their possible applications to design theory.

At a more practical level, application of the theory to real design environments has the potential to improve communications between designers by providing a common vocabulary, to assist in the standardization of design specifications, and to lead to new and more powerful software tools to aid the designer. The latter avenue is explored in the next Part of this document.

Part IV

ENGINEERING COMPUTATION

Chapter 12

Introduction

The work presented thus far has been quite abstract. This has been necessary because little fundamental work exists in the area. Because of its abstract nature, practical applications of the author's work may seem difficult to identify. In order to address this problem, the author has included a major application: the impact of HM on engineering computing. The intention is to extend the perspective afforded by HM into more practical issues of computation in design environments. In particular, a software system developed by the author, called DESIGNER, will be presented and discussed as a demonstration of the gains that can be expected from the application of formal techniques.

The influence of computer technology in design has become a strong one. Computers and computer software systems are used in every aspect of the engineering enterprise, from concurrent engineering, to CAD and analysis, simulation, manufacturing and production, and marketing. Although some may argue that this dependence is somehow detrimental to the profession of engineering, there is little doubt that computer technology represents the only currently available, feasible means by which to control the complexity of the engineering enterprise.

The Information Revolution the world is currently experiencing is related to its predecessor, the Industrial Revolution. "By dramatically reducing the costs of coordination and increasing its speed and quality, these new [information] technologies will enable people to coordinate more effectively, to do much more coordination and to form new coordination-intensive business structures [18]." Coordination, in this case, is the ability to organize, or *order*, what we do, and the information we use to do it. One of the principal

gains we may expect, then, from formal theories like HM is the ability to generate software systems far more capable than existing ones to assist designers in their work. Indeed, successful software systems based on formal theories can provide a strong indication of the suitability of the theories themselves.

Currently available programming and database systems are insufficient for engineering applications [32, 42]. The high degree of complexity of modern design artifacts, the richness of the number and kinds of interrelationships in information, and the imprecision with which terms (such as *design intent* and *functional description*) are used have all contributed to this problem. Today's continuing and ever-increasing rate of new engineering software development makes it evident that these issues have not yet been addressed properly.

As well, there have been a number of arguments made recently in favor of a "language-based approach" to design (e.g. [162, 175]). Such an approach entails the development of languages that permit the expression of design information in a computational environment that do not tie the user to a single methodology or solution technique. In this way, relatively small languages can permit the capture of relatively large quantities of information, while permitting the users a certain freedom of expression that would be missing if a precise methodology had been specified. The author's approach throughout the work presented herein has been to separate information from procedures that use the information, at both the theoretical and practical levels. The language-based approach, then, is quite sympathetic to the author's approach, and the programming language to be described in this Part may be regarded as a contribution to such language-based research in design.

The inherent complexity and relatively *ad-hoc* nature of the design enterprise can overwhelm even the most powerful systems. In order to combat this, engineering software developers and researchers have rightly searched for methodologies emphasizing organization and ordering of information: increasing the degree of organization in a collection of information makes it more accessible, more concise and less likely to be a source of error for designers. However, the exact nature of the organizational forms that would best suit the engineering enterprise in general have to date eluded discovery.

One problem not addressed in other current research in this area is that of general engineering computing [173]. Not since the creation of Fortran has a language been targeted intentionally for use by engineers (with the possible exception of Ada, whose success in this area has yet to be proven). Considering the great evolution that has occurred in design, it is not surprising that Fortran is unable to meet the

programming requirements of modern engineering enterprises. Given the high degree of complexity of design information, it is important that the paradigm match the conceptual model of the users.

In order to motivate the arguments presented in this work, we shall begin by examining the relationship between software systems and engineering in an abstract way. This will permit us to distinguish between the general exigencies of engineering computing and the requirements of particular software systems.

The *domain* of a software system is the collection of the kinds of problems to which the system can be applied. The software system maintains an internal computational model of its domain; that is, the system is a manifestation of this model. This domain-specific computational model is a unification of a *formal* conceptual model plus a general model of computation. For example, continuum mechanics and mathematics provide a formal model for finite element analysis; this, combined with a programming environment (which is a manifestation of a general model of computation), provides the computational model for finite element software systems.

Engineers use a software system to solve problems within the system's domain; they are using the system's model of the problem domain to perform a given task. In order to do this, they must at least have some understanding of the system's computational model. This includes the formal model and also (unfortunately) some aspects of the general computational model.

However, the users also have their own mental, or *cognitive*, models of the problem domain. If the user's model of the domain is incompatible with either the formal domain model inherent in the software system or the computational model of the software system itself, an *impedance mismatch* [27] results. As the name implies, an impedance mismatch represents an efficiency loss in information modeling. In this sense, the word "efficiency" refers to the ability of a system (formal *or* software) to precisely and correctly model necessary information in a timely manner with respect to the user's model. The greater the mismatch, the less usable the software system becomes, and the more likely inefficiencies and errors will dominate its use.

Impedance mismatches between the author's formal domain model (HM) and the user's cognitive model are handled by the isomorphism described in Part III. The isomorphism provides a correspondence between the formal system and design information that effectively eliminates impedance mismatches arising from the use of HM in a design environment. In this Part, the author is concerned primarily with controlling impedance mismatches between the *computational* models of engineering software systems

and the user's cognitive model.

Impedance mismatches are not *bugs*. Bugs are simple mechanical errors remedied in a relatively straightforward way. Rather, impedance mismatches arise from limitations in the programming paradigm selected for the design and implementation of the software system. These mismatches are problematic. They define from the outset the limits of the software system's functionality, and guide the programmers throughout the development of the system. The paradigm provides the conceptual framework for general computation within which the software will be created. Any limitations in this framework will be inherited by the software. If the selected paradigm is incompatible with the formal model of the problem domain or the users' cognitive model, the resulting software will be clumsy and error-prone, and its results will be difficult to interpret and use. But most importantly, there is no simple solution to this kind of impedance mismatch short of choosing another programming paradigm and rewriting the software from scratch.

Clearly, then, the investigation and selection of programming paradigms is the key to controlling mismatches and improving the quality of software systems. Engineering researchers, as experts in the problem domain for which suitable software systems are to be developed, are likely candidates to be able to carry out such investigations successfully.

The author expects that successful efforts will bring to existence entirely new programming paradigms evolved from existing ones, but significantly different in the computational models they support. These models will be based on formal theories about design that are currently under development by many different researchers, thus providing continuity of formal rigor from the domain model through to the implementation of support software. The design theories will provide the necessary formal background, and the computational models will provide the bridge from the purely theoretic world to the software systems that will be able to address real engineering problems.

In the Sections that follow, a programming language for design information, called DESIGNER, will be introduced and discussed. DESIGNER is an implementation of a computational model based on HM, and must thus satisfy its axioms. Since HM describes the structure of design information without making statements about the use of that information, DESIGNER will deal with *static* data modeling only; that is, it will deal with the representation of information, *nor* with its manipulation. Chapter 13 presents the requirements for a new computational paradigm for engineering. Based on these requirements, three programming paradigms are identified as the bases of the current work. Chapter 14 then introduces

certain key notions and the basic syntactic forms of the DESIGNER language. Chapter 15 follows up with several examples indicating how the DESIGNER language can capture design information effectively and concisely.

Chapter 13

Requirements for a New Programming Paradigm

Many different programming paradigms have been used to develop engineering software, including functional, logic, imperative, object-oriented, and relational [176–178]; none have succeeded universally. The latest attempts, and those showing the most promise, are those that blend two or more paradigms. This is because each paradigm alone represents a particular *projection* of the "real" world to that of computation; that is, each paradigm performs only certain classes of computations very well, and in their purest forms do so *to the exclusion* of other kinds of computation. Although there are many problem domains where one paradigm alone can perform well, engineering, for all its complexity, is not one of them. Engineering problems can cover both numeric and geometric domains, both the precisely analytic and the numerically approximated, the ethereal (for example, using chaos theory to model turbulent fluid flow) and the practical (manufacturing process planning, amongst others). Clearly, an appropriate solution for the engineering enterprise must perform computations efficiently, yet remain flexible enough to meet its widely divergent requirements.

Some of the requirements of a new programming paradigm for engineering have already been mentioned. To reiterate, a candidate paradigm must: (a) be supported by a formal model of the application domain; (b) capture the complex data structures typical of many engineering applications; (c) capture and manage the rich interrelationships that exist between these structures; (d) be representable by a formal computational

model; and (c) eliminate, or at least minimize, impedance mismatches. The author considers one other goal as well: the paradigm should provide an environment for continued research in engineering computing and design theory.

It is noted that due to the current lack of formalism in engineering design, it is difficult to expect the immediate satisfaction of some of these requirements. We can, however, use such formalization as currently exists, and tailor our results in the future as our understanding deepens. Thus flexibility becomes another goal of the paradigm. In this context, flexibility is the ability to try new programming and modeling techniques easily without significant alterations to existing software. We may in this way substitute our lack of formal understanding of design with empirical analysis and the ability to experiment.

The author has identified three programming paradigms as being of particular relevance in engineering applications: functional programming, semantic data modeling and object-orientation. We now briefly introduce these paradigms.

13.1 Functional Programming

The fundamental paradigm of functional programming is that a computation can be represented by a collection of (possibly complex) operations acting on (relatively simple) data structures. In the ideal case, no assignments ever occur; that is, there are no variables in functional programs, only functions and constants (e.g. numbers, strings, etc.). Though this approach is very elegant from a theoretical point of view, it does not take into account the exigencies of practical computation; for example, the integer `10` is represented in a purely functional environment as the application of a function called `successor` to the integer `0` ten times. In a real programming environment, the computational overhead of such an approach would be completely unacceptable. Thus, most functional languages permit at least *single assignment*, i.e. a variable can be assigned a value only once during its lifetime. One does not change the value of a variable, but rather eliminates it in favor of an entirely new variable that has the new value assigned to it. This approach, though an acceptable compromise between the practical and theoretical aspects of computation, results in a computational model that is rather counter-intuitive and difficult to use, especially by people unaware of the theory of functional programming. To offset this, many currently popular functional languages (particularly LISP and Scheme) permit multiple assignment, but only in

certain controlled ways. Although it is commonly said that the functional paradigm does not permit the capture of *state*, this is a misconception. Functional programming only requires that manipulation of the state of a program be *explicit*. This ensures that unwanted and often difficult to identify *side-effects* do not occur in functional programs. References [176, 179, 180] discuss various aspects of functional languages, and [181] discusses some of the advantages of using the functional paradigm in engineering applications.

One of the most novel ideas of the LISP family of functional languages is that of *meta-circularity*. A meta-circular language is defined largely in terms of itself, turning programs written in that language into extensions of the language itself. This permits the creation of self-modifying programs and active data structures. Meta-circularity also unifies the roles of programmer and user: it becomes easier for the "programmer" to understand the user's needs and for the "user" to have control over software programmed by others. It also permits the development of software in modular layers, all written in the same language, each layer expanding upon the capabilities of the preceding layers.

The ability of these languages to treat programs as data (i.e. perform symbolic computation [179]) is of particular advantage in applications involving design information modeling. We can write functions that can examine design models and check for satisfaction of various kinds of constraints. The constraints may be specific to the domain (e.g. constraining weight or size of a component in an assembly), but they may also be computational in nature (e.g. database integrity constraints, normalization, etc.). We can thus unite database schema and instances of those databases (i.e. models of actual design artifacts), treating them with a single language and computational paradigm. This further integrates the roles, and thus blurs the distinction, between programmer and user.

The greatest advantage of functional programming is the sound and detailed theoretical background upon which it rests [99, 182]. Firstly, the λ-*calculus* presents a general mathematical theory of functions, and is based on set theory. Secondly, *denotational semantics* provides a notation (based on the λ-calculus) for the representation and study of computation, permitting the evaluation and manipulation of programs as if they were algebraic expressions. Such formal computational theories provide the bridge between a formal design theory, and the implementation of software systems based on that theory.

Although Scheme and LISP are rather similar, Scheme [183–185] was chosen as the base language for DESIGNER because its semantics is defined in much more formal terms than LISP. This means Scheme exhibits a higher degree of robustness and formal rigor than its predecessor. A particular implementation

of Scheme, Elk [186], was used because because it provides certain implementation features that made the development of DESIGNER quicker and easier in a UNIX[1] environment. It is noted, however, that DESIGNER has also been ported to another version of Scheme, called SCM[2], which is more rigorously adherent to the existing Scheme standard [184] than is Elk. This indicates conclusively that DESIGNER is a legitimate extension of Scheme.

Finally, it is noteworthy that there has been some effort [181, 187] to study and implement the notion of a database based on the functional paradigm. The significant advantage of the functional paradigm in this area is that the *explicit* management of program state can make database update management and normalization easier than in conventional approaches.

The one notable disadvantage of the functional programming paradigm is that complex data structures are normally not supported. This restricts the organizational structures that can be imposed on design information. However, as will be demonstrated below, this deficiency can be remedied by the use of concepts from object-orientation. Although recently developed functional languages (e.g. ML, Haskell) have richer type systems[3] and data structures, it is not yet clear how these mechanisms can be best used in an engineering environment. As well, the type systems supported by these languages are not necessarily compatible with the formal design theories they are intended to support. The author therefore favors the use of an untyped language (like Scheme) into which can be built whatever type information is found necessary.

13.2 Semantic Data Modeling

Semantic data modeling was originally conceived of to permit the creation and description of data schema that would then be coded into relational implementations [111]. It presented the advantage of ignoring implementation issues in favor of achieving a deeper conceptual understanding of the problem domain. Implementation issues were then dealt with using the relational data model [108]. Eventually, it was recognized that there were application domains where the richer assortment of abstraction mechanisms

[1] UNIX is a trademark of AT&T.

[2] SCM is maintained by Aubrey Jaffer, copyright ©1989 Free Software Foundation, Inc.

[3] In computational theory, *type systems* deal with the specification of kinds of data structures (e.g. integers, character strings, records, procedures, etc.). This is not to be confused with the use of the term "type" in HM.

available in semantic modeling would be of great advantage. Engineering was identified as one such domain. Since then, a number of different semantic data models have been generated and implemented. Two good surveys of the field are [28, 111].

Because of the diversity of approaches taken by various researchers in semantic data modeling, it is difficult to identify key notions representative of the general approach. However, one very relevant notion has found use in the definition and implementation of DESIGNER, and is worthy of note here. Attributes of data structures in semantic data models are regarded as functions mapping one (group of) object(s) to another (group of) object(s). This is noticeably different from the view taken in object-orientation (see below), wherein attributes are commonly seen as constituent parts of data objects and are thus considered more structural than procedural. For example, given some objects modeling engines and some other objects modeling fuels, semantic modeling would define an attribute `uses-fuel` as a function mapping engine objects to fuel objects; alternatively, in object-orientation, an attribute `fuel` would be a component of engine objects whose values are chosen from available fuel objects.

The author has found that the semantic modeling view of attributes is quite useful in implementing an object-oriented system within a functional language (i.e. DESIGNER). This is particularly interesting because it indicates a relationship between two very different computational paradigms (functional and object-oriented programming) by way of a data modeling system not originally intended for such a purpose. Exactly how attributes are dealt with in DESIGNER will be discussed in the Sections to follow.

13.3 Object-Orientation

Object-orientation has had as its goal, since its origins in the language Simula, the modeling of entities of interest to the user in a very high-level and thus usable manner [188]; in other words, object-orientation permits the creation of software models of "real-world" entities that are very similar to the users' mental models of those entities. By definition, then, object-orientation has at least the potential of minimizing impedance mismatches in applications where the information to be modeled must be presented and manipulated in as straight-forward a manner as possible [27], such as engineering design.

An *object* is generally defined as an entity that can capture all relevant information about a particular real-world entity. Objects *encapsulate* their implementation, thus making their usage entirely independent

of how they function internally. This frees the user from having to understand the details of how the data structures operate. Encapsulation has been shown to provide many advantages in the development and use of software from the points of view of both computer science [159,188–190] and engineering [25,112,191,192].

Communication between an object and another object or a user occurs by passing *messages* between objects; a message is a request that an operation be carried out by an object or group of objects. Objects can be active (dispatching messages to each other) passive (requiring an external agent for message dispatch), or some mix of the two. Similar kinds of objects can be grouped together in various ways, permitting different organizations of collections of objects that reflect the requirements of the problem domain. These *abstraction mechanisms* are particularly important in a domain like engineering, where the nature of relationships between data can be both very complex and very important.

The greatest advantage of object-oriented systems is that they permit the modeling of complex, highly interrelated entities (such as those found in an engineering environment) in a more simple, flexible and elegant way than can other programming paradigms. The principal disadvantage of object-oriented systems is that there is still no consensus on exactly what the nature of objects should be: there are no formal models for computing with objects. It should be noted, however, that there are a number of on-going research efforts aimed at providing a more formal footing for object-orientation (e.g. [193–195]) both as a programming paradigm and as a database model.

The author has identified three parameters that individuate object-oriented technologies. These parameters are discussed here insofar as selection of alternative approaches based on them helped form the overall structure of DESIGNER.

13.3.1 Message Passing Protocols

The first parameter deals with protocols for message passing. A recent report by the Object Oriented Database Task Group of ANSI [196] differentiates between two kinds of message passing protocols. The first, *classical* message passing, is similar to that provided by languages like SMALLTALK-80 [197] and C++ [198]. The protocol defines a particular object as the recipient of the message. The message contains a *selector*, which is used by the receiver to identify a suitable procedure (called a *method*) to be invoked.

Methods are defined within object *classes* (see below). Other arguments to the method may also be provided by the message. The method is evaluated with respect to the receiver of the message and the result is returned to the sender of the message.

The second form of message passing is called generalized or *canonical* message passing. In this protocol, a message takes the form of a function call. There is no explicitly defined receiver; rather, all objects are passed as arguments and treated equally. The result of the evaluation of the message is returned to whatever environment originally dispatched it.

The classical message passing protocol permits methods to be associated only with those objects for which evaluation of the message is meaningful. Depending on the approach taken to define methods, this can improve encapsulation and is thus advantageous. However, it can also introduce asymmetry in the way actions on objects are regarded by the user.

For example, in SMALLTALK-80, the message `3 + 4` is evaluated as follows: the object identified as `3` is sent the expression `+ 4`; a method associated with the appropriate class of objects is identified based on the receiver (in this case, integer addition) and evaluated by taking the value of the argument (`4`) and adding it to the value of the receiver. Now, if we had previously defined `x ← 3` and `y ← 4`, evaluation of the message `x + y` would result in `x` taking the value `7`, while `y`'s value remained `4`. The asymmetry is manifested in the different roles played by the two terms `x` and `y`: although they would conventionally be considered equivalent in that they both simply represent values, classical message passing causes the role of `x` to be active and that of `y` to be passive. This asymmetry can become confusing, especially in more complicated cases typical in design. Furthermore, classical message passing causes *side-effects*; that is, the change in the value of `x` is an *implicit* change in the state of the program that is *uncontrollable* by the user. The existence of side-effects can prevent verification of software models of design entities; they are thus detrimental to the construction of reliable software systems.

On the other hand, the canonical message passing protocol would evaluate `x + y` by using the function `+` to create a new object whose value is the sum of the arguments. This form eliminates the asymmetry and appeals to the intuition more than the classical form. The function is not bound directly to a type or class of object. Though this may be seen as a violation of encapsulation (i.e. the function is not defined within an object class), techniques exist to offset this loss while maintaining the advantages of the classical form. These techniques will be discussed below in conjunction with the description of DESIGNER.

Finally, the canonical message passing protocol eliminates the requirement for the special meta-variable *self*, which refers to the receiver of a message. In the classical protocol, methods are attached to object classes; there is no way to determine *a-priori* which particular object will have to evaluate a given method in response to a message. Therefore, there is no *a-priori* way to identify the receiver of a message. To circumvent this problem, researchers use a special variable commonly named *self* which, at any point in a computation, identifies the receiver of the currently active method. The problem is that *self* is an inherently self-referential structure, and makes verification of programs and software models very difficult. Since the canonical message passing protocol does not explicitly associate methods with objects or object classes, there is no need for this special variable, and we are spared a great deal of complexity.

13.3.2 Abstraction Mechanisms

The second issue regards the abstraction mechanisms used to group objects. There are two abstraction mechanisms currently competing as the principal mechanism for this purpose: *classes* and *prototypes*. They are both based on grouping objects by similarities in their properties. Generally known as "classification" or "typing", this mechanism is also a basis of human cognitive function in general, and is recognized as such by psychologists, philosophers, and artificial intelligence researchers [28, 131, 166].

Classes are the older of the two forms, and are used in such languages as SMALLTALK-80 and C++. They define the structure and behavior of collections of objects. Classes are more abstract entities than are objects: the latter describe real entities to be modeled, whereas the former describe the objects themselves. They are commonly dealt with using the same syntactic forms as plain objects (*instances*), but have significantly different semantics.

Prototypes, on the other hand, are plain objects that are used as templates to create other objects. They are entirely different from classes because classes are more abstract modeling entities than objects, whereas *any* object may be a prototype. Thus, the same syntax *and* semantics can be used uniformly throughout prototype-based systems.

Prototype-based languages have the potential to replace class-based systems as the standard for object-oriented programming because prototype-based computational models are simpler than class-based ones, but retain the full expressive power of the latter [199].

There are problems with class-based systems that are dealt with better by prototypes, especially in engineering computing. Firstly, schema – or meta-data – evolution is orthogonal to the development of objects in class-based systems, because classes and their instance objects are of different degrees of abstraction. In prototype-based systems, there is no orthogonality. This simplification can be of great assistance especially in engineering, where relationships are already complex without the added requirements of different orthogonal systems for data and meta-data. Secondly, version control, which is very important in maintaining accurate histories of designs, is greatly simplified in prototype-based systems. Thirdly, one-of-a-kind modeling of design artifacts is much more straight-forward using prototypes, since there is no need to generate classes for which there will ever be only single instances. SELF [200] is one system that has been demonstrated to provide potentially all of the benefits of classes while maintaining the higher flexibility associated with prototypes.

13.3.3 Hierarchical Construction of Objects

The final issue that distinguishes various object-oriented languages is the manner by which hierarchies of objects are constructed. Again, there are two principal alternatives. In *inheritance*, given a message, the corresponding method is searched for and used by the receiver in the evaluation process. The criterion upon which the search process is based depends on prescribed relations between various object classes. One class inherits from a *super*-class if its instances respond to all the messages to which instances of the super-class also respond. In other words, the receiver of a message in inheritance-based systems evaluates methods that have been located in other objects.

Delegation, on the other hand, can be viewed as message transformation: given a certain message, the delegation constructs and transmits a new message based on the given one. The new message replaces the original and is re-dispatched in its place. Thus, in delegation-based systems, the receiver object is sent as an additional argument to whatever object has the method used to evaluate the message; this is the converse of what happens in inheritance-based systems. In general, inheritance is used in class-based languages and delegation is used in prototype-based languages. Though both these approaches have been used in various language implementations to date, neither has shown marked advantages over the other.

13.4 Combining Object-Orientation and Functional Programming

Although functional programming and object-oriented programming are often considered to be at opposite ends of the language spectrum, it is interesting to note that many functional languages have had object-based or object-oriented extensions for some time. Scheme (and LISP) have both been used to generate object-oriented systems [201–206], but almost all these systems have been class-based. Some systems (e.g. [207]) have supported prototypes rather than, or in addition to, classes, but none of these efforts capture the intention of a system that would be useful for design. They are general purpose programming languages or platforms for language research, and too primitive to be applied to formal design theories such as HM.

It may be argued that a distinct similarity exists between the theoretical notions of an object and a *closure* [208]. A closure is a data structure often used in functional programming to represent the conjunction of a function with an environment, providing values for any variables or identifiers not otherwise defined within the function itself. Both objects and closures are capable of capturing the *state* of a computation. Although this similarity is quite evident, it is not sufficient to completely unify the two programming paradigms. However, it is a good sign, and indicative of possibilities for success.

One distinct difference between functional and object-oriented programming that must be reconciled in any attempt to merge the two is the issue of static versus dynamic scope. In static scoping, values for variables may be identified by static lexical analysis of the program text. This means that the environment used to evaluate a given function is the environment that was current when the function was *defined*. In dynamically scoped systems, the environment used to evaluate a function is that which exists when the function is *called*. Arguably, functional programming is at its best in a statically scoped system [176], whereas, since the first version of SMALLTALK-80, object-oriented languages have favored dynamic scoping. Various approaches have been suggested in the literature [176, 202, 209]; a definitive solution to the problem does not appear to exist yet. The author's approach takes advantage of multiple assignment in Scheme to provide the minimal dynamic scope needed to support encapsulation of state. It is interesting to note, however, that the first object-oriented language, Simula, used static rather than dynamic scope [209].

13.5 Summary

This Chapter has introduced the requirements for a new programming paradigm for design, and identified three key existing paradigms that can be used to meet these requirements. The two principal requirements are: that the paradigm model design information in as direct a manner as possible (so as to eliminate or at least minimize impedance mismatches), and that formal rigor be maintained as far as possible between the formal domain model (HM) and the implementation of the programming paradigm as a real computerized tool. Object-orientation provides the modeling constructs needed to meet the first requirement, while functional programming provides the degree of formalization needed to meet the second requirement. Also, notions from semantic data modeling provide a unique, expressive and efficient mechanism for the treatment of attributes (as defined in HM) within a computational framework.

Chapter 14

Concepts and Forms in Designer

This Chapter introduces the key notions and syntactic forms of DESIGNER, a prototype-based object-oriented language implemented in Scheme. The primary goal of this exposition is to demonstrate (a) how object-oriented and functional programming can be effectively combined, and (b) that there are significant advantages in the use of formal tools to design and build software systems for engineering applications.

Only the principal DESIGNER forms will be introduced in this Chapter. Other, ancillary forms will be introduced in the next Chapter as required for the various examples.

In order to avoid circularity within the definition of DESIGNER, a distinction is made between actions *of* objects, and actions *on* objects. This distinction prevents the definition of DESIGNER from being self-referential. Since DESIGNER is meant to satisfy HM, which is not self-referential, it too must not be self-referential.

Actions *of* objects are the operations they are meant to carry out. Requests for such operations take the form of *messages*. Actions *on* objects occur at a different degree of abstraction; in DESIGNER, these actions are captured by regular Scheme functions.

14.1 Syntactic Conventions

Since DESIGNER is an extension of Scheme, it obeys Scheme's syntactic rules. Scheme is similar to LISP in that its fundamental structure is a list, denoted by values enclosed in parentheses. Function calls take the form of lists whose first element is the name of the function. For example: `(+ 3 4)` is a function call to +, with arguments 3 and 4. In the general case, we write a kind of statement (i.e. a *form*) in DESIGNER using the format:

```
(FUNCTION-NAME ARGUMENT ARGUMENT...)
```

where capitalized words represent the denotation intended of data in those relative positions within the call. An ellipsis (...) indicates zero or more items of the same kind as that immediately preceding it. Thus we could generalize `(+ 4 5)` as:

```
(ARITH-OP NUMBER...)
```

where `ARITH-OP` stands for any arithmetic operator and `NUMBER...` stands for a sequence of at least one number[1].

Comments in Scheme are started by a semicolon and continue to the end of the line.

14.2 Creating Objects

Objects in DESIGNER are representations of objects as defined in HM. A DESIGNER object is a passive store of attributes; this treatment is in keeping with the selected message-passing mechanism (see Section 14.6, below), the general philosophy of the functional paradigm, and the definition of objects in HM.

There are three ways to create new objects in DESIGNER. The first is through the use of `gen-object`:

```
(gen-object)
```

[1] In Scheme, arithmetic operators can take any number of arguments; for example `(+ 2)` simply returns 2, and `(+ 3 4 5 6)` returns 18.

This is the most primitive object-forming function. It returns a new, empty data structure representing an object. The object has no attributes, will respond to no messages, and is in no way related to any other object. At the user's level, this function is of little use; however, it is at the heart of all other more sophisticated object-forming functions.

An object is implemented as a vector[2] containing (a) a list of the object's attributes, (b) a list of the object's parents, and (c) a list of constraint relationships between attributes in that object. The notions of *parents* and *constraints* will be discussed below.

Secondly, existing objects can be *cloned* to form new ones using the form:

```
(clone OBJECT)
```

The argument of `clone` may be any object. A clone has all the attributes of its *parent*, the object from which it was cloned. The parent is the prototype for the new object. The attributes of the new object are automatically initialized to have values equal to the values of the parent's attributes. The parent/child relationship between an object and its clones is similar to the relationship between classes and instances in languages like SMALLTALK-80, in that the parent provides the information needed to define the structure of the child. Cloning is generally not intended for the user, but can be utilized in creating hierarchies of objects and quickly creating copies of objects.

The third way to create objects is with the *new* form:

```
(new PARENT-SPEC ATTR-SPEC INIT-SPEC)
```

Because DESIGNER is a prototype-based language without the notion of object classes, *new* provides the functionality to instantiate existing prototypes as well as the functionality to create new prototypes. It replaces both the class instantiation and subclassing mechanisms in conventional object-oriented languages. It is noted that *new* satisfies the axiom of specialization in HM.

`PARENT-SPEC` is either a single object or a parenthesized list of objects that will be the parents of the new object. The set of attributes of the new object is the cartesian product of the attribute sets of its

[2]Scheme vectors are fixed-length sequences whose values are indexed much as one-dimensional arrays are accessed. Furthermore, the elements of a vector may be arbitrary Scheme objects, including number, strings, functions, etc.

parents. Since an object can have more than one parent, DESIGNER provides a kind of *multiple inheritance*. However, the axiom of specialization in HM restricts objects to be formed only by a union operation of disjoint parent objects; that is, the parents of an object cannot share attributes.

Other attributes not available in any of the parent objects can be defined by means of `ATTR-SPEC`, a list of attribute specifications. If no extra attributes are needed to define a new object, an empty list, `()`, must appear as the second argument of `new`. The exact syntax of attribute specifications is described in the next Section.

Finally, `INIT-SPEC` is a sequence of forms providing initial attribute values. The syntax of an `INIT-SPEC` form is:

```
(ATTRIBUTE-NAME INITIAL-VALUE)
```

where `ATTRIBUTE-NAME` is the symbol by which a particular attribute is identified, and `INITIAL-VALUE` is a DESIGNER form that is evaluated to provide the initial value of that attribute. Only those attributes for which default values are insufficient need be initialized; default values are taken from the parents of the new object.

All constraints are checked before a final value is returned from `new`.

Finally, some general remarks regarding these forms are in order.

Identity of objects is based on the corresponding axiom in HM (axiom 3). To be exact, two objects are identical if the sets of attributes of the objects correspond, and if the values of the members of each corresponding pair of attributes are equal.

All the object-forming functions described above permit at most the addition of attributes to objects; no facility is currently provided for the removal of attributes.

A clone keeps track of all of its parents. The sequence containing all parents of an object, all parents of the parents, and so on, is called the *lineage* of an object. Any object occurring in the lineage of an object is referred to as an *ancestor* of the object. Information regarding parents of objects is important for two reasons. First, it is used to create attributes in cloned objects, thus defining object *state*. Second, it permits sharing procedural information between objects of similar kinds: since an object knows what objects it descends from, it can behave like its parents.

The author has found that a prototype-based approach can also capture the conventional notion of object *classes*. In fact, the definition of a class-like object using only the low-level functions provided by DESIGNER's prototype system has been successfully implemented. This supports the notion that the prototype-based approach is at once more general than, yet as equally expressive as, the class-based approach of languages like SMALLTALK-80.

14.3 Attributes

Attributes in DESIGNER are constrained values stored in objects. The constraints limiting the types of values an attribute can have are referred to as *domain constraints*. Examples of domain constraints include `integer?` and `Cuboid?` which are unary predicates that return True only if their arguments are integers or "Cuboids", respectively. The domain of an attribute in DESIGNER is the set of values that satisfies the attribute's domain constraint. Attributes are identified by a name unique among all the attributes of an object.

Attributes are added to objects when they are created, using the `new` form introduced above. Attributes are defined within `new` by giving a specification of its component parts using the syntax:

```
(NAME DOMAIN INITIAL-VALUE)
```

where `NAME` is the name by which the attribute shall be identified , `DOMAIN` is a unary predicate defining the domain constraint, and `INITIAL-VALUE` is an optional initial value. If the initial value is omitted, the attribute is given the value `no-val`, a special symbol in DESIGNER indicating no assignment has been made. `no-val` satisfies any constraint in DESIGNER, and is intended to differentiate between object creation and assignment of values to object attributes. Such a distinction is important because a user will often know that a certain object will be used, but may not *a-priori* know exactly what the specific nature of the object will be [2].

When an attribute is defined, DESIGNER automatically defines a *query function*, used to query the attribute for its value, and a *setter function*, used to set its value. The syntax of these two kinds of functions is given as:

```
(QUERY-FN OBJECT)
```

```
(SETTER-FN OBJECT VALUE)
```

A query function takes an object as its single argument and returns the current value of the corresponding attribute in that object. A setter function takes an object and a value as its arguments and assigns the value to the corresponding attribute in that object. Assignment occurs if the domain constraint is satisfied; if a domain constraint violation occurs, an error message is displayed and no assignment occurs.

The names of query and setter functions are based on the name of the attribute. For example, creating an attribute named `length` causes the creation of a query function named `?length` and a setter function named `length:`. `?length` and `length:` will function correctly for all objects with an attribute named `length`, and are created only once. This convention does not clash with Scheme's convention of naming predicates with a trailing question mark, yet provides sufficient connotation to make the meaning of these functions clear.

The query and setter functions of an attribute are representative of the attribute itself; the implementation of attributes in DESIGNER is not visible to the user. From the user's point of view, these functions *are* the attribute. Thus we are taking advantage of the semantic data model's notion of *attributes* as functional mappings between objects.

We note finally that the setter function makes use of the `set!` primitive in Scheme to alter the state of an attribute in an object. The author feels justified in this "corruption" of the functional paradigm insofar as attribute state is strongly encapsulated within objects; the side-effects are therefore controlled by the semantics of the setter function and beyond the reach of other objects and the user.

14.4 Constraints between Attributes

Domain constraints on single attributes are defined via domain predicates. Constraints can also be imposed between many attributes in a given object. These constraints are referred to as *object constraints*, and are defined using the following form:

```
(constrain OBJECT ((NAME...) FORM...) ...)
```

where OBJECT is the object to be constrained, (NAME...) is a list of the attribute names involved in the constraint, and the remaining forms FORM... are the body of the constraint, the evaluation of which should return a boolean value. To facilitate adding many constraints to a single object, more than one constraint can be given in each `constrain` form.

Object constraints in DESIGNER are passive structures. That is, if a constraint is evaluated and found to be not satisfied, DESIGNER will signal this fact to the user, but will not attempt to alter the model in order to satisfy the constraint. The author reasons that the responsibility of such alterations should fall on either the user or some software system of a higher level than DESIGNER (e.g. a expert system, neural net, or other knowledge-based controlling system). DESIGNER is meant only to capture design information statically; it is not a goal-oriented manipulatory system. Put another way, DESIGNER is intended to answer the question *What is this design?* rather than *How was this design created?*

A particular object constraint is checked whenever an assignment is attempted to any attribute involved in the constraint. However, the assignment occurs whether the object constraint is satisfied or not. This is acceptable due to the exigencies of design. It is often the case that in the course of a design process, various constraints may not be satisfied. This is quite normal in design [2] and does not necessarily mean the design is inadequate. It may be that the constraints or the design problem itself are not well-posed. That is, in design, the model of the artifact *evolves* as design proceeds, and it is normal for there to be instants when the model is inconsistent. These are not, and should not be treated as, fatal errors, but rather as intermediate steps.

14.5 Function Overloading

In Scheme, one may create and use a function without assigning the function a particular name. Such functions are called *anonymous* functions. Overloading of function names is achieved in DESIGNER by grouping a number of anonymous functions in a list accessible only to a special, *generic* function. Various definitions of the terms "overloading" and "generic functions" have been suggested in the literature (e.g. [197, 198, 210, 211]). Here, we use the term "overloading" to denote a language symbol that has multiple, non-exclusive definitions, and "generic function" to denote a function that can operate successfully on a number of different types of objects, where object types are defined by their ancestry.

In order to distinguish between different anonymous functions bound to the same generic function, each anonymous function has associated with it a *signature* indicating the number of parameters required by the anonymous function and a (possibly complex) predicate that is true only for sets of actual parameters that are acceptable for that anonymous function. The generic function is bound to a user-supplied name. When a generic function is called (by referring to its name), the actual parameters are compared by the system to the signatures attached to each anonymous function. The first anonymous function with a signature matching the actual parameters is applied to those parameters, and the resulting value is returned.

Generic functions are created in DESIGNER using the `overload` form:

```
(overload NAME ((ID...) (PREDICATE...) FORM...) ...)
```

where `NAME` is the symbol to be overloaded as a generic function, `(ID...)` is a list of formal parameters to be used in the definition of a particular anonymous function, and `(PREDICATE...)` is a list of predicates that, together with the list of parameters `(ID...)`, form the signature of the anonymous function. `FORM...` is the body of the anonymous function. To facilitate writing many overloadings of a single symbol at once, `overload` can accept many anonymous function definitions.

By convention, names of generic functions begin with a colon (e.g. `:volume`).

Generic functions combined with object lineage information permit the sharing of functions that act on objects (i.e. *methods* in conventional object-oriented languages). A signature may contain a predicate that checks for membership of an argument in the lineage of an object. This is equivalent to the IS_A relationship in HM and implies that the child object *inherits* from its parent. Thus, we can cause different behaviors in generic functions depending on the ancestry of its arguments. Examples are given in the next Section that demonstrate how this mechanism is similar to polymorphism in conventional object-oriented languages.

14.6 Message Forms

This author has found that the canonical message passing protocol lends itself well to implementation in a functional environment, especially if generic functions are used. It also maintains a single, consistent syntactic convention for the expression of both function calls and messages. Although classical message

passing had been implemented in earlier versions of DESIGNER, the canonical form is found to be easier to implement, much more efficient, and has led to a much more usable system.

The canonical form diminishes the importance of inheritance and/or delegation. This is because it establishes methods to exist outside objects rather than to be part of them. Nonetheless, the ultimate goal of inheritance and delegation -- the sharing of behavior (actions) between objects without requiring the existence of object classes or other higher-order structures -- is achieved.

14.7 Intentional Versus Extensional Attributes

In DESIGNER, we distinguish between extensional and intentional attributes. An *extensional* attribute is one that has an actual value assigned to it, whereas the value of an *intensional* attribute is derived from the values of other attributes. Extensional attributes are static; that is, their values are not procedural. They are stored within objects themselves and are representative of the *state* of the object. Intentional attributes, however, are procedural (active) in nature. Messages implemented with generic functions capture intentional attributes. Examples of this are given in the next Section.

The author notes that it is not always obvious whether a particular property of a design entity should be modeled with an extensional attribute or an intentional attribute. The criteria for making this decision are bound up in the requirements of the design process used, and can vary widely. For example, if the weight of a particular component is to be constrained, it may be preferable to model the component's dimensions extensionally, and its volume intentionally. However, if the constraint is based on a restriction of space that the component may take up, then it may be preferable to model its volume extensionally and its dimensions intentionally (e.g. as parameterized functions of the volume). This indicates a relationship between constraint and attribute specifications. An in-depth study of this relationship is deferred for future work.

14.8 Summary

In this Chapter, the author has introduced the principal notions and syntactic forms of DESIGNER, a computer programming language that combines object-orientation in a functional programming environment

such that the axioms of HM are satisfied. In this way, continuity of formal rigor is maintained from the logical aspects of HM through to the implementation of DESIGNER itself. Though unconventional in some regards, the DESIGNER language provides a numberof mechanisms for the direct modeling of design information in a concise manner.

Chapter 15

Examples

DESIGNER is an extension of the Scheme programming language, providing a prototype-based object system. A signature-based, canonical message-passing mechanism permits overloading of function names. Thus, DESIGNER objects satisfy the axioms of HM while also providing certain very convenient programming mechanisms to increase usability and efficiency.

In addition to the actual object-oriented extensions of Scheme, DESIGNER also includes a library of generally useful objects. The library can be expanded by the user. Information used to generate some of the algorithms implemented in DESIGNER and in the prototype library was taken from [212–215]. The complete source of DESIGNER and of the prototype library is given in the Appendices.

15.1 Simple Examples

A simple example of the definition of two object prototypes is given in Figure 15.1.

The first statement in the example defines `Cuboid` to be an object. It is generated by *specialising* a single object, `Object`, defined within DESIGNER as the root parent of all other objects. The statement is not a message because it acts *on* objects. It is taken as a convention in DESIGNER that objects intended to be used as prototypes have capitalized names (e.g. `Object`, `Cuboid`) while other objects have names consisting entirely of lower-case letters. This rule is not enforced by the syntax of DESIGNER, but is used

```
(define Cuboid
        (new Object
             ((x number-gt0? 1)
              (y number-gt0? 1)
              (z number-gt0? 1))))
(make-type-predicate Cuboid)

(define Sphere
        (new Object
             ((radius number-gt0? 1))))
(make-type-predicate Sphere)

(overload :volume ((s) ((Sphere? s)) (* (/ 4 3) Pi (** (?radius s) 3)))
                  ((c) ((Cuboid? c)) (* (?x c) (?y c) (?z c)))))
```

Figure 15.1: Definition of two Objects.

only to improve readability.

Three extensional attributes are then added to `Cuboid`, named `x`, `y` and `z`, representing the dimensions of the entity. All three attributes are constrained to be numbers greater than zero by the domain specification `number-gt0?` (assuming we accept a physical dimension to be positive and non-zero). A default value of `1` is assigned to each attribute.

The second statement, `(make-type-predicate Cuboid)`, creates a predicate `Cuboid?` that takes a single argument and returns True or False, depending on whether the argument is an object and an ancestor of `Cuboid`.

In the example, a `Sphere` object is also defined, as well as a predicate `Sphere?`. `Sphere` has one attribute representing a radius.

Next, the symbol `:volume` is overloaded for both `Cuboids` and `Spheres`. It accepts a single argument descended from either `Cuboid` or `Sphere`, calculates the volume of its argument, and returns this value. This means that any object cloned from `Cuboid` or `Sphere` is a valid argument. `:volume` is an example of an intentional attribute; that is, an attribute defined as a function of other attributes. Representing intentional attributes as functions assures that the values of these attributes will always reflect the most recent values of the attributes upon which they depend.

Domain constraints on extensional attributes are checked only when new values are about to be assigned to the attributes, since assignment is the only operation that can change their values. If the new value does

not satisfy these constraints, the assignment does not occur and an error is signalled.

Constraints may also be imposed between attributes in an object. For example, if we wished to constrain the `Cuboid` object such that its y dimension is always twice its x dimension, we could write a constraint as in Figure 15.2.

```
(constrain Cuboid ((x y) (= y (* 2 x))))
```

Figure 15.2: A constraint on Cuboid.

This object constraint will automatically be checked each time an attempt is made to assign values to either the x or y dimensions of `Cuboid` or any object cloned from `Cuboid`. However, as discussed in the previous Section, violation of object constraints does not prevent assignment from occurring.

Constraints on intentional attributes are represented differently. Since intentional attributes do not have explicitly defined values and are represented by functions of objects, constraints on these attributes occur at a different level of abstraction. For example, if there were a circumstance that required constraining a `Cuboid`-like object so that its volume were not to exceed 100 units, we could use forms similar to those in Figure 15.3.

```
(define Thing
        (new Object
             ((box Cuboid?)
              (max-vol number? 100))))

(constrain Thing ((box max-vol) (<= (volume box) max-vol)))
```

Figure 15.3: Example of constraint on intentional attributes.

This restriction on the formation of constraints on intentional attributes is reasonable because such constraints arise from the interaction of objects (design entities) with their environment, i.e. other objects with which they interact. In the example, the maximum volume is not a component of our model of cuboids. It is inappropriate, then, to embed such constraints within an object when they in fact model relationships between (possibly) many objects.

At any time, the constraints on an object may be checked with the function `check-object-constraints`, which takes an object as its single argument, and returns a boolean value indicating whether the constraints

on the object's attributes are satisfied or not.

```
(define box (new Cuboid () (x 3) (y 6) (z 5)))
(?x box)                                  ⇒ 3
(:volume box)                             ⇒ 90
(define ball (new Sphere (r 5)))
(:volume ball)                            ⇒ 523.5987756
```

Figure 15.4: Examples of DESIGNER queries and messages.

Figure 15.4 shows examples of DESIGNER queries and messages. The text following the ⇒ symbol shows the return value when these statements are evaluated. The first statement defines a new `Cuboid` object whose dimensions are initialized to 3, 6 and 5 respectively.

The second statement in Figure 15.4 is `(?x box)`. This is a query for the value of the attribute called `x` in object `box`. It returns the value 3.

The third statement shows the retrieval of the current value of an intentional attribute of `box`: its volume. This is actually a message, though in form it appears the same as a function call. Because its single argument is a descendant of `Cuboid`, the anonymous function within `:volume` that returns the volume of `Cuboid` objects is evaluated.

It is noted that `Cuboid` and `box` are both objects, and that there is no essential difference between them. In particular, either one may be used as a prototype for the generation of other objects.

We then create a new `Sphere` object, named `ball`, with a radius of 5 units. The `volume` message is then evaluated again. But because its argument descends from `Sphere` this time, a different anonymous function is evaluated.

DESIGNER's ability to define functions that act differently depending on their arguments is similar to polymorphism and dynamic binding in class-based, object-oriented languages. In the terminology of these more conventional languages, we would say that `volume` is a message accepted by instances of both `Sphere` and `Cuboid` classes, but implemented by different methods in each class.

15.2 Multiple Inheritance

DESIGNER supports a form of multiple inheritance. Objects may inherit from more than one parent object, but none of the parents may share attributes; that is, the sets of attributes contributed by each parent must be disjoint from the sets of attributes of all other parents. This restriction is imposed by HM to maintain its logical validity. Other object-oriented languages have attempted to support multiple inheritance of overlapping parents or classes, but none has succeeded completely. Indeed, the author considers the problematic nature of managing overlapping parents and classes in inheritance to be a clear indication of inconsistency, and prefers to restrict the possible kinds of relationships that can be specified for the sake of formal rigor.

In order to introduce the notion of multiple inheritance as supported by DESIGNER, we include a small example based on an example in [210, 216] which deals with the conceptual modeling of vehicles, machines, and automobiles. Figure 15.5 gives both a graphical depiction of the model and the DESIGNER forms used to implement the appropriate prototypes.

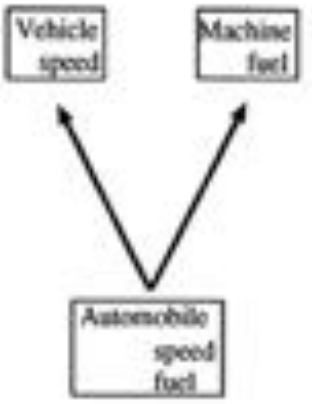
In the example, `Automobile` inherits from both `Vehicle` and `Machine`. The last three forms are queries demonstrating that the attributes of both parents have been passed on to `Automobile`.

Although one may question the validity of modeling an automobile as in this example, it is used here only to demonstrate the straight-forward nature of the multiple inheritance mechanism provided by DESIGNER. The validity of using multiple inheritance as in this example is in itself worthy of study; however, we defer such discussion since it does not bear directly on the subject at hand.

15.3 Mimicry of Classes

Another example indicating the flexibility of DESIGNER involves the generation of a class-like object as might be found in other object-oriented languages (such as SMALLTALK-80). The implementation of this object requires the use of the low-level functions of DESIGNER and is less than 70 lines of code. A simple example of its use for queue objects (first-in, last-out lists) is given in Figure 15.6.

The example demonstrates that the distinction is made between classes (e.g. `Queue`) and instances (e.g. `q`).




```
(define Vehicle
        (new Object
             ((speed number? 0))))

(define Machine
        (new Object
             ((fuel symbol? no-val))))

(define Automobile
        (new (Vehicle Machine)))
(fuel: Automobile 'gasoline)

(?speed Automobile)                    ⇒ 0
(?fuel Automobile)                     ⇒ gasoline
(?fuel Machine)                        ⇒ no-val
```

Figure 15.5: An example of multiple inheritance in DESIGNER.

```
(new-class Queue () (lst () '()))

(overload push ((q v) ((Queue? q)) (lst: q (cons v (?lst q)))))
(overload pop ((q) ((Queue? q))
                   (if (not (?lst q)) (error 'pop "queue is empty."))
                   (let ((r (car (?lst q))))
                      (lst: q (cdr (?lst q)))
                      r)))
(overload as-list ((q) ((Queue? q)) (?lst q)))

(class? Queue)                          ⇒ true
(instance-variables Queue)              ⇒ (lst ())

(define q (instance Queue))
(instance? q)                           ⇒ true
(class? q)                              ⇒ false
(is-a q)                                ⇒ Queue

(push q 1)
(push q 2)
(push q 3)

(as-list q)                             ⇒ (1 2 3)
```

Figure 15.6: Example of use of DESIGNER Class object.

and that the important relation `is-a` correctly returns the class of an instance. Also, in the `new-class` form, the first empty parentheses are for a list of (possibly many) superclasses, thus providing support for multiple inheritance. In this particular example, no superclasses are specified.

15.4 Preliminary Design/Synthesis – Four-Bar Linkage

This Section describes a more complex example of the kinematic synthesis of a four-bar linkage given three precision points. The three point synthesis technique used here is taken from [34], pages 103-110.

Figure 15.7 presents a schematic representation of the geometry of the four-bar linkage with various objects labelled, and a single link with its various parts labelled. The objects themselves are discussed below.

The four-bar linkage is modeled as an object (`4bar`, see Figure 15.8) with five attributes. `base-a` and `base-b` are the base connections of the driver and output links respectively. The attributes `input`,

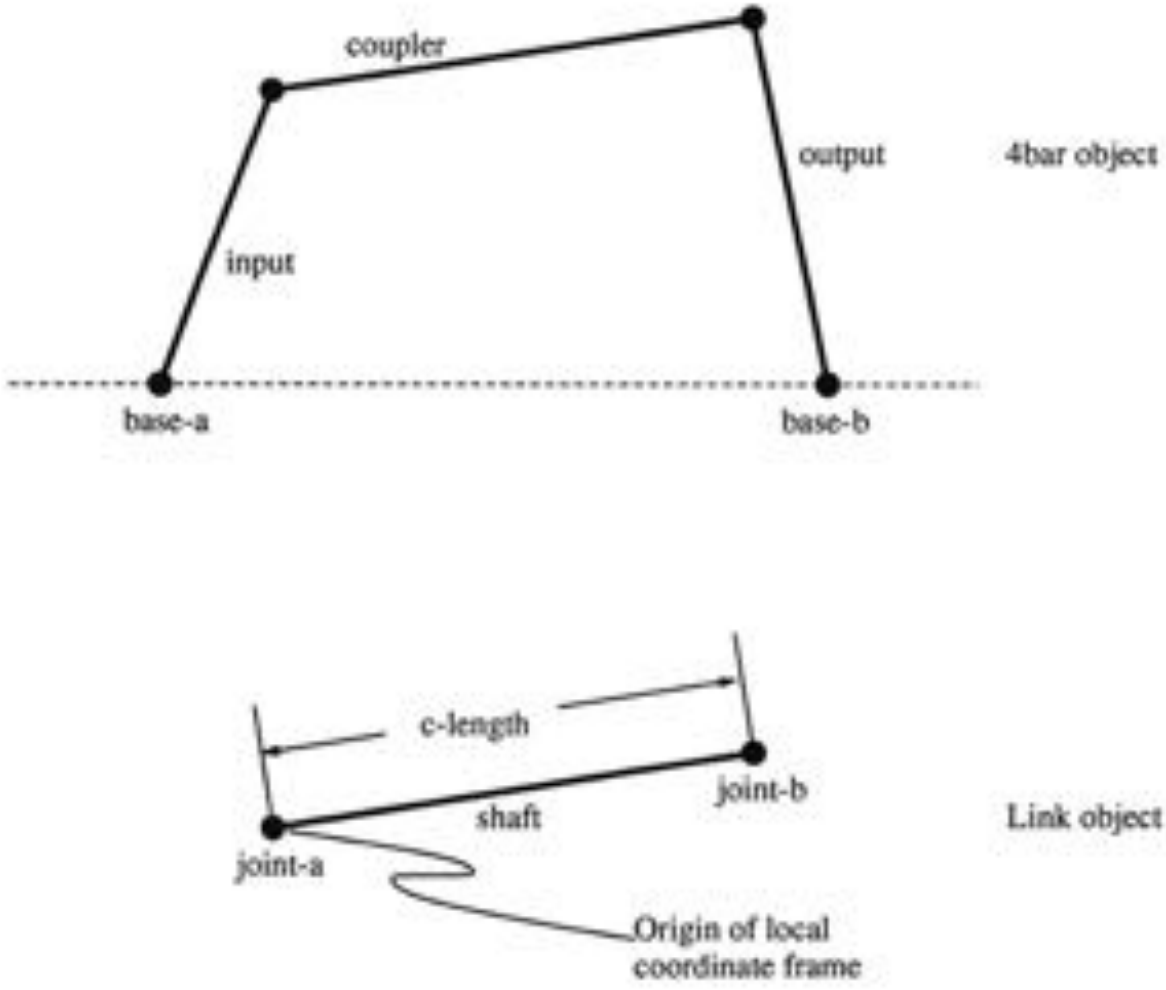


Figure 15.7: Schematic geometry of 4bar and Link objects.

coupler and output model the three moving links of the mechanism.

```
(define 4bar
  (new Part
       ((base-a Coord?)
        (base-b Coord?)
        (input Link?)
        (coupler Link?)
        (output Link?))))
(make-type-predicate 4bar)
```

Figure 15.8: Definition of four-bar linkage object.

Our definition of 4bar depends on three other objects: Coord, Part and Link. These objects are prototypes defined in the DESIGNER prototype library. Coord models 3D points. Part represents mechanical parts; descendants of Part may be components or assemblies, and are aggregates of other objects modeling a physical part's geometric and physical properties (i.e. material type, etc.). Link is a specialization of Part specific to the development of the four-bar linkage model, and is defined in Figure 15.9.

```
(define Link
  (new Part
       ((c-length number-gt0? 1)
        (shaft Part?)
        (joint-a Part?)
        (joint-b Part?))))
(make-type-predicate Link)
```

Figure 15.9: Definition of Link objects.

The geometry of Link is a complex one which could be based on a solid model. Its attributes include a shaft and two joints. The joints are used to connect links to one another. In addition, Link has one other attribute: a characteristic length that represents the distance from one joint of the link to the other. It is used as a constraint on the geometry of the link and is generated as part of the solution of the three-point synthesis technique. The values of the shaft and joint attributes are no-val by default; as the actual geometry of individual links is defined, we can add constraints to Link that will assure that the geometric properties of the shaft and joints maintain a relationship defined from the synthesis method via the link's characteristic length (see Figure 15.10).

```
(constrain Link ((c-length shaft joint-a joint-b)
                 (:= c-length
                     (:- ((:intersect (?axis joint-b) (?axis shaft))
                          (:intersect (?axis joint-a) (?axis shaft)))))))

(make-constructor Link (cl) (new Link () (c-length cl)))
```

Figure 15.10: Geometric constraints for `Link` objects.

The definition of `Link` introduces a new DESIGNER function, `make-constructor`. This function is meant for convenience and creates a *constructor* function that facilitates the instantiation of prototypes. In this case, `make-constructor` will create a function `@Link` that creates a new `Link` object and uses its single argument to initialize the link's characteristic length. Many, though not all, prototypes in DESIGNER have constructors defined for them.

The three-point synthesis technique is implemented in DESIGNER as a method activated by the message `(3pt-synthesis ...)` and is given in Figure 15.11.

A detailed explanation of the `3pt-synthesis` method is unnecessary here. Essentially, the technique uses various angular and linear displacements of the four-bar linkage through the three prescribed points to generate enough information to create a new `4bar` object that satisfies the input parameters. A series of examples taken from [34] were used to test the algorithm; our results were numerically identical to those given in the reference. The heart of the method is the last form, `(new 4bar ...)`, which actually creates a clone of `4bar` and returns it.

The author notes that the `3pt-synthesis` method is not covered explicitly by HM itself because it is a *procedural* component of design; it is included here as a vehicle by which design models created with DESIGNER can be directly manipulated to perform useful design operations.

This representation of a four-bar linkage is a parameterized model suitable for a number of purposes. If, for example, a kinematic analysis of the four-bar linkage were to be performed, the `4bar` object could be extended to capture the information needed to constrain the components of the four-bar linkage assembly. These constraints can be specified in DESIGNER with the forms in Figure 15.12.

The base points `base-a` and `base-b` are not considered part of the geometry of the linkage itself, but rather components of constraints placed on parts of the geometry. Specifically, the ends of the input

```
(overload 3pt-synthesis
    ((D2 D3 phi2 phi3 gamma2 gamma3 psi2 psi3)

     ((Complex? D2) (Complex? D3) (number? phi2) (number? phi3)
      (number? gamma2) (number? gamma3) (number? psi2) (number? psi3))

     (let ((Cphi2 (@Complex (cos phi2) (sin phi2)))
           (Cphi3 (@Complex (cos phi3) (sin phi3)))
           (Cgamma2 (@Complex (cos gamma2) (sin gamma2)))
           (Cgamma3 (@Complex (cos gamma3) (sin gamma3)))
           (Cpsi2 (@Complex (cos psi2) (sin psi2)))
           (Cpsi3 (@Complex (cos psi3) (sin psi3)))
           (Z1 no-val) (Z2 no-val) (Z3 no-val)
           (Z4 no-val) (Z5 no-val) (Z6 no-val))

       (set! Z1 (/ (- (* D2 (- Cgamma3 1)) (* (- Cgamma2 1) D3))
                   (- (* (- Cphi2 1) (- Cgamma3 1)) (* (- Cgamma2 1) (- Cphi3 1))))))
       (set! Z2 (/ (- (* (- Cphi2 1) D3) (* D2 (- Cphi3 1)))
                   (- (* (- Cphi2 1) (- Cgamma3 1)) (* (- Cgamma2 1) (- Cphi3 1))))))
       (set! Z3 (/ (- (* D2 (- Cgamma3 1)) (* (- Cgamma2 1) D3))
                   (- (* (- Cpsi2 1) (- Cgamma3 1)) (* (- Cgamma2 1) (- Cpsi3 1))))))
       (set! Z4 (/ (- (* (- Cpsi2 1) D3) (* D2 (- Cpsi3 1)))
                   (- (* (- Cpsi2 1) (- Cgamma3 1)) (* (- Cgamma2 1) (- Cpsi3 1))))))
       (set! Z5 (- Z2 Z4))
       (set! Z6 (- (+ Z1 Z5) Z3))

       (new 4bar (base-a (@Coord 0 0 0))
                 (base-b (@Coord (?real Z6) (?imag Z6) (z 0))
                 (input (@Link (magnitude Z1))
                 (coupler (@Link (magnitude Z5))
                 (output (@Link (magnitude Z3))))))))))
```

Figure 15.11: Three point synthesis method.

```
(constrain 4bar
           ((base-a input)
            (:locate input (lambda (p o)
                             (and (= (?x p) (?x base-a))
                                  (= (?y p) (?y base-a))
                                  (= (?z p) (?z base-a))))))
           ((input coupler)
            (:locate coupler (lambda (p o)
                               (and (= (?x p) (?c-length input))
                                    (= (?y p) 0) (= (?z p) 0)))
                     input))
           ((coupler output)
            (:locate output (lambda (p o)
                              (and (= (?x p) (?c-length coupler))
                                   (= (?y p) 0) (= (?z p) 0)))
                     coupler))
           ((output base-b)
            (:locate output (lambda (p o)
                              (and (= (?x p) (?x base-b))
                                   (= (?y p) (?y base-b))
                                   (= (?z p) (?z base-b))))))))
```

Figure 15.12: Kinematic constraints for `4bar` objects.

and output links of the four-bar linkage are constrained to remain at the coordinates defined by the base points, though they are free to rotate, and successive links in the linkage are similarly constrained. In the constraint specification, `base-a`, `base-b`, `input`, `coupler` and `output` refer to attributes in a `4bar` object; `:locate` is an overloaded message to any `Part` that takes two or three arguments. The first argument is an attribute whose value is to be spatially constrained. The second argument is a function definition (a *lambda* form) that specifies the nature of the constraint, and takes two arguments: the position and orientation of the attribute to be constrained. The optional third argument is another attribute in whose coordinate frame the spatial constraint is to occur. The representation of spatial coordinates with respect to non-global coordinate frames is reminiscent of the relative coordinate formulation of variational solid modeling taken by Fogle [75]; in that work, it is demonstrated that the use of relative coordinate frames can simplify the specification of spatial relationships. The current work with DESIGNER appears to corroborate Fogle's findings.

The first constraint is between the base point `base-a` and the `input` link: `:locate` is used to constrain the position of `input` to map exactly to the position of `base-a`. The second constraint is between the `input` and `coupler` links, and constrains the origin of `coupler` to be at the end of `input` opposite

where `input` is attached to `base-a`. In this case, the constraint is defined with respect to the coordinate system of `input`. The other constraints are similarly defined.

The `4bar` object defined herein models a four-bar linkage that satisfies the input data at an abstract, conceptual level. Although the actual shape of the links has not been defined, their one essential property, their characteristic length, has been captured. The value of these attributes become constraints on the actual geometries of the links.

This example demonstrates that DESIGNER is capable of more than just modeling design artifacts themselves; it provides the means to capture information about the *entire* design. For example, the `3pt-synthesis` method defines a relationship between the functional requirements of a four-bar linkage, and the key design parameters that define the physical solution. This indicates that DESIGNER can represent the relationship between functional requirements of a design problem specification and the physical parameters that define a solution. Specifications expressed as generic functions acting on objects define relationships between the requirements of a design artifact and the objects themselves capture key design parameters that define the physical solution. This indicates that DESIGNER can represent the relationship between functional requirements of a design problem specification and the physical parameters that define a solution.

15.5 Hierarchical Organization – Thermal Analysis of a Wall

This final example shall focus on the organization of design information with DESIGNER. Specifically, we shall present (a) a parameterized model of a wall, and (b) a representation of steady-state heat flow for the wall model. This example is inspired by the material in [97]. It is noted here at the outset that the model presented below is not the *only* way one could represent a wall in DESIGNER; this particular model was chosen because (a) it matched the author's cognitive model of walls and (b) it is sufficient for demonstration purposes in this document.

Throughout the following text, the reader may refer to Figure 15.13; this figure depicts graphically the structure of the wall model, including all objects as well as all inheritance and aggregation relationships.

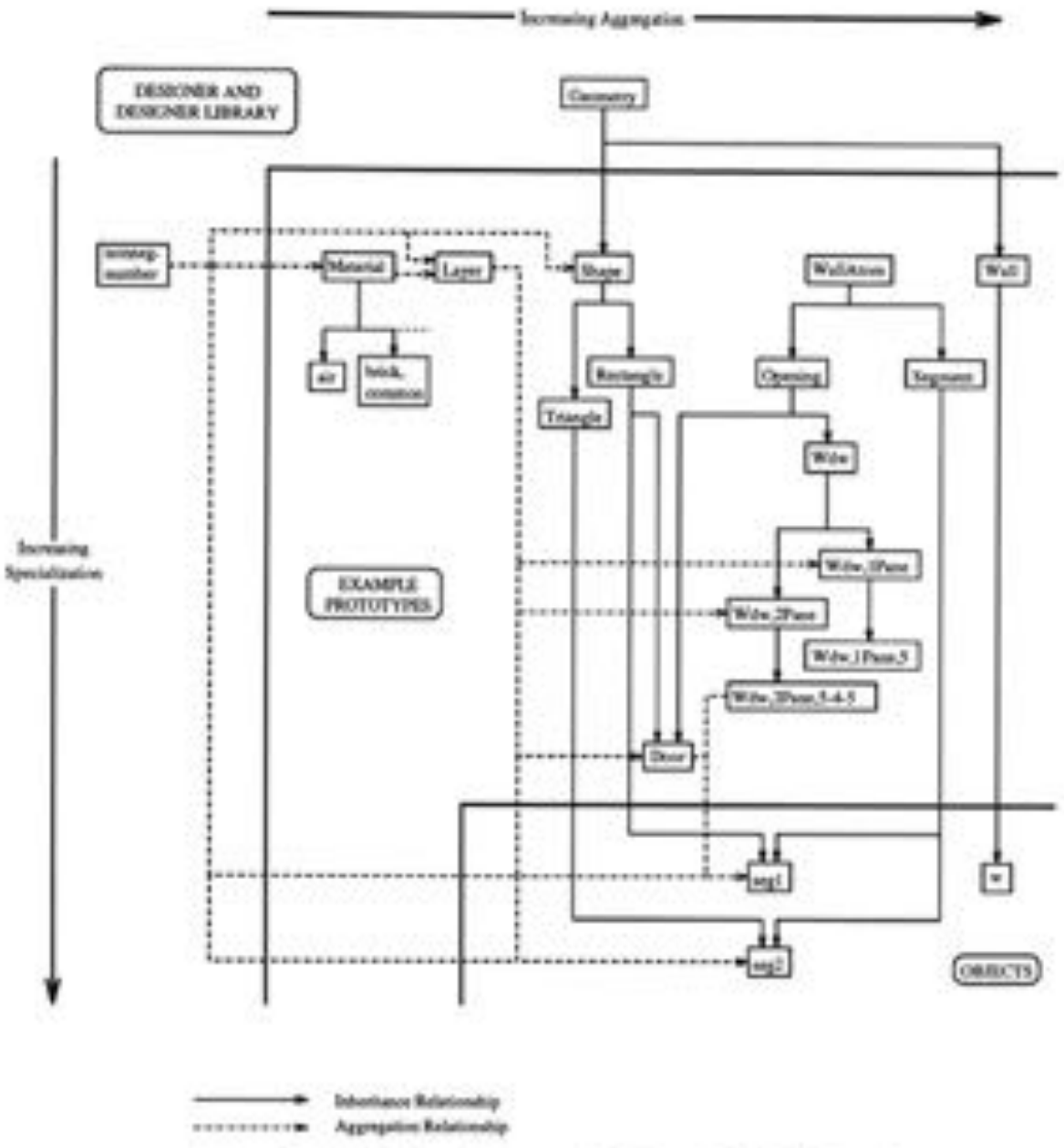


Figure 15.13: Inheritance/Aggregation Network for Wall Example.

15.5.1 Structural Modeling Considerations

There are two important aspects that must be considered to generate a useful model of a wall: its geometry and its composition.

The width and height of a wall are important in defining its relationship to other structural elements, but are not essentially tied to its composition[1]. The constraints on wall width and height exist at the level of assemblages of many walls and thus are beyond our single-wall model; so, from the point of view of this example, height and width are arbitrarily defined values.

The thickness of a wall, however, must be treated differently because it depends on the wall's composition. A wall is composed of various layers, each serving a specific purpose – load bearing, insulation, covering, and so on. Each layer is constrained according to the requirements of that particular wall, which in turn constrains the wall's overall thickness. So though the height and width of a wall are arbitrary (from the point of view of the wall model), its thickness is not.

Therefore, we will represent height and width as purely geometric extensional attributes and thickness as an intentional attribute depending on the wall's composition.

15.5.2 Thermal Analysis Modeling Considerations

For the thermal analysis portion of this example, we will make use of the following physical relationships (drawn from [97] and a standard thermodynamics text [214]).

The heat flow through a wall is represented approximately by:

$$\dot{Q} = \frac{kA}{t}\Delta T \tag{15.1}$$

where ΔT is the change in temperature through the wall from the warmer side to the cooler side, t and A are the thickness and area of the wall respectively, and k is the thermal conductivity of the wall.

Furthermore, by analogy with electrical systems, we may define an overall coefficient of thermal conduc-

[1] For stress analysis and structural integrity, we would likely need to constrain the size of the wall based on its composition: it must be able to carry its own weight. However, we are only interested in thermal analysis for this example.

tivity, U, for a wall of many layers of constant area as:

$$U = \frac{1}{\sum_i r_i} \tag{15.2}$$

where r_i is the thermal *resistance* of the ith layer of the wall, defined by:

$$r_i = \frac{l_i}{k_i}. \tag{15.3}$$

15.5.3 Definition of Wall Prototype Objects

We begin by defining a prototype for simple planar geometric `Shape` objects (Figure 15.14), with attributes of `height` and `width`. `Shape` objects will be used to define the major components of walls. The area of these shapes will be of importance in the thermal analysis, so we include the definition of an intentional attribute `:area`. `Shape` inherits from `Geometry`, an object defined in the DESIGNER prototype library which models an arbitrary spatial object, providing a local coordinate frame for the object and messages to perform various transformations.

```
(make-type-predicate Shape)                    ; abstract prototype
(define Shape
  (new Geometry
       ((width number-gt0? 1)
        (height number-gt0? 1))))

(make-type-predicate Triangle)                 ; subtype for triangles
(define Triangle (clone Shape))

(make-type-predicate Rectangle)                ; subtype for rectangles
(define Rectangle (clone Shape))

(overload :area
          ((t) ((Triangle? t)) (* (?width t) (?height t) 0.5))
          ((r) ((Rectangle? r)) (* (?width r) (?height r)))
```

Figure 15.14: 2D Shape objects.

Next, we need to represent the notion of a layer of a wall. We shall assume that a single layer is composed of a single material and is of constant thickness. `Layer` objects are defined in Figure 15.15. For the

sake of the thermal analysis, we include a message definition for `:therm-resist`, which returns the thermal resistance of a layer of a given thickness and material.

```
(make-type-predicate Layer)
(define Layer
  (new Object
       ((material Material?)
        (thickness number-gt0? 1))))
(make-constructor Layer (m t) (new Layer () (material m) (thickness t)))

(overload :therm-resistance
          ((l) ((Layer? l)) (/ (?thickness l) (?therm-cond (?material l))))))
```

Figure 15.15: Layer objects.

The `Material` prototype is defined in Figure 15.16; for brevity, we have included only one necessary property, thermal conductivity, and the few instances needed for this example. Thermal conductivity data was taken from [214]. Numeric values are in SI units.

```
(make-type-predicate Material)
(define Material
  (new Object ((therm-cond number-gt0? 1))))

(define brick.common (new Material () (therm-cond 0.69)))
(define brick.face (new Material () (therm-cond 1.31)))
(define glass.window (new Material () (therm-cond 0.78)))
(define plaster.gypsum (new Material () (therm-cond 0.48)))
(define wood.pine.yellow (new Material () (therm-cond 0.147)))
(define wood.pine.white (new Material () (therm-cond 0.112)))
(define wool.rock (new Material () (therm-cond 0.039)))
(define air (new Material () (therm-cond 0.02624))) ; at 300K
```

Figure 15.16: Material Prototype and Instances.

In order to permit the creation of walls with more complex geometries than those described by `Shape` objects, we define a *segment* to be an area-wise component of a wall. Segments will be defined with `Shape` and `Layer` objects. We will also be able to use segments to define walls that have regions composed of different layers. However, before we define a prototype to represent wall segments, we must consider one other compositional element of walls: *openings*. An opening is intended to generalize the notion of passages through a wall. For this example, we will only consider doors and windows. An opening exhibits the same properties as segments: they occupy a certain area, and are composed

of possibly many layers (e.g. multi-paned windows). Since it is desirable to minimize the information content of our model, we will begin by defining a `WallAtom` object that will capture those properties common to both wall openings and segments (see Figure 15.17). It is noted that none of these prototypes will use the `Shape` prototype defined above for planar shapes. `WallAtom` and its descendants are used to capture information specific to walls other than their shape. We will, however, make use of the `Shape` prototype later.

```
(make-type-predicate WallAtom)
(define WallAtom (clone Object))

(overload :thickness
          ((a) ((WallAtom? a))
               (apply + (foreach-attribute a Layer? ?thickness))))

(overload :therm-cond
          ((a) ((WallAtom? a))
               (/ 1.0
                  (apply + (foreach-attribute a Layer? :therm-resistance)))))
```

Figure 15.17: Atomic wall components for openings and segments.

The `:therm-cond` message calculates the overall coefficient of thermal conductivity of a `WallAtom` according to the mathematical model in Section 15.5.2. The instances of `foreach-attribute` gathers all attributes in `WallAtom` objects that are `Layers`, applies a function to them (`?thickness` and `:therm-resistance` respectively), and returns the list containing the results of the function applications.

Now we can use `WallAtom` to define prototypes for wall openings and segments. The only real distinction between openings and segments is that segments may contain openings, but openings cannot contain other openings (i.e. a window cannot have another window as a component).

```
(make-type-predicate Opening)
(define Opening (clone WallAtom))

(overload :spec-heat-flow
          ((o) ((Opening? o)) (* (:therm-cond o) (:area o)))))
```

Figure 15.18: Prototype for wall openings.

First, we define an `Opening` object in Figure 15.18. The *specific heat flow* of an `Opening` is de-

fined as the rate of heat flow per degree of temperature, and is represented by the intentional attribute `:spec-heat-flow`.

```
(make-type-predicate Segment)
(define Segment (clone WallAtom))

(overload :opening-area
          ((s) ((Segment? s))
               (apply + (foreach-attribute s Opening? :area)))))

;;; wall segment area = total area - opening area
(overload :segment-area
          ((s) ((Segment? s)) (- (:area s) (:opening-area s)))))

(overload :spec-heat-flow
          ((s) ((Segment? s))
               (+ (apply + (foreach-attribute s Opening? :spec-heat-flow))
                  (* (:therm-cond s) (:segment-area s))))))

(overload :heat-flow
          ((s dt) ((Segment? s) (number? dt))
                  (* (:spec-heat-flow s) dt))))
```

Figure 15.19: Prototype for wall segments.

Second, we define a `Segment` object in Figure 15.19. In this case, we differentiate between the total area of the wall segment, the area of all the openings (represented by the intentional attribute `:opening-area`) and the area of actual wall material (represented by the attribute `:segment-area`). The specific heat flow (`:spec-heat-flow`) of a segment is the sum of the specific heat flows of each opening and the specific heat flow of the rest of the wall. We finally define the attribute `:heat-flow` to calculate the actual heat flow through a given `Segment` for a given temperature difference.

Next, we shall specialize `Opening` for both doors and windows. First, we define `Door` in Figure 15.20. `Door` inherits multiply from both `Opening` and `Rectangle`: the former provides those aspects that represent design intent and function (at least, insofar as thermal analysis is concerned) as well as composition, whereas the latter provides those aspects representing its other geometric characteristics. The `@Door` constructor (created by `make-constructor`) simplifies the creation of single-layered doors. If some particular door has more than one layer (unlikely though that might be), it can be created using `new`.

Various prototypes are created for different kinds of windows in Figure 15.21.

```
(make-type-predicate Door)
(define Door (new (Opening Rectangle)
                  ((layer Layer?))))
(make-constructor Door (m w h t)
                  (new Door ()
                       (layer (new Layer () (material m) (thickness t)))
                       (width w)
                       (height h)))
```

Figure 15.20: Prototype object for doors.

```
(make-type-predicate Wdw)
(define Wdw (clone Opening))

(make-type-predicate Wdw,1Pane)
(define Wdw,1Pane
  (new Wdw
       ((pane Layer? (new Layer () (material glass,window))))))
(make-constructor Wdw,1Pane (t)
                  (let ((w (clone Wdw,1Pane)))
                    (thickness: (?pane w) t)
                    w))

(make-type-predicate Wdw,2Pane)
(define Wdw,2Pane
  (new Wdw
       ((pane1 Layer? (new Layer () (material glass,window)))
        (gap Layer? (new Layer () (material air)))
        (pane2 Layer? (new Layer () (material glass,window))))))
(make-constructor Wdw,2Pane (t-pane1 t-gap t-pane2)
                  (let ((w (clone Wdw,2Pane)))
                    (thickness: (?pane1 w) t-pane1)
                    (thickness: (?gap w) t-gap)
                    (thickness: (?pane2 w) t-pane2)
                    w))

(make-type-predicate Wdw,1Pane.5)
(define Wdw,1Pane.5 (##Wdw,1Pane 0.005))

(make-type-predicate Wdw,2Pane.5-4-5)
(define Wdw,2Pane.5-5-5 (##Wdw,2Pane 0.005 0.004 0.005))
```

Figure 15.21: Prototype objects for windows.

We begin by defining a simple `Wdw` object, specialized from `Opening`. `Wdw` is then specialized into single-paned (`Wdw.1Pane`) and double-paned (`Wdw.2Pane`) types. In both cases, glass is used as the material for the panes, and in the case of `Wdw.2Pane`, the interstitial space contains air. Constructors (`0Wdw.1Pane` and `0Wdw.2Pane`) are created for convenience. Finally, in the last four lines of Figure 15.21, two specific kinds of windows are created: `Wdw.1Pane.5`, a single-paned window with a 5 millimeter pane of glass; and `Wdw.2Pane.5-4-5`, a double-paned window with two 5 millimeter panes and a 4 millimeter air gap between them.

One last prototype needs to be defined: the wall itself. Since all the important functions for thermal analysis have been defined within `Segment` and `Opening`, the `Wall` object need not be much more than an aggregate used to gather together various `Segment` objects. `Wall` is defined in Figure 15.22. Our model provides intentional attributes for the overall area of a wall (`:area`) and the total area of all openings in a wall (`:opening-area`). It also provides a message `:heat-flow` that calculates the total heat flow through a wall for a given temperature difference.

```
(make-type-predicate Wall)
(define Wall (clone Geometry))

(overload :area
          ((w) ((Wall? w)) (apply + (foreach-attribute w Segment? :area)))

(overload :opening-area
          ((w) ((Wall? w))
               (apply + (foreach-attribute w Segment? :opening-area))))

(overload :heat-flow
          ((w dt) ((Wall? w) (number? dt))
                  (* (apply + (foreach-attribute w Segment? :spec-heat-flow))
                     dt)))
```

Figure 15.22: Prototype object for walls.

15.5.4 Example of Wall Model Usage

We now present an example of the use of these prototypes to define a particular wall, and calculate the heat flow through it. The sample wall will consist of two segments, a large rectangular segment containing a window and a door, and a smaller triangular segment with no openings. Figure 15.23 defines these two

segments, and the wall they compose.

```
(define seg1
  (new (Segment Rectangle)                    ; Segment #1
       ((door Door? (@Door wood,pine,white 1 2 0.04))
        (wdw Wdw? (new (Wdw,1Pane,5 Rectangle) ()
                       (width 1) (height 0.5)))
        (outer Layer? (@Layer brick,face 0.1))
        (core Layer? (@Layer wool,rock 0.1))
        (inner Layer? (@Layer plaster,gypsum 0.01)))
       (width 4) (height 3.5)))

(define seg2                                  ; Segment #2
  (new (Segment Triangle)
       ((outer2 Layer? (@Layer wood,pine,white 0.005))
        (outer Layer? (@Layer brick,common 0.1))
        (core Layer? (@Layer wool,rock 0.1))
        (inner Layer? (@Layer wood,pine,yellow 0.005)))
       (width 4) (height 1.5)))

(define w                                     ; The wall itself
  (new Wall
       ((s1 Segment? seg1)
        (s2 Segment? seg2))))
```

Figure 15.23: Two sample wall segments.

The rectangular segment, `seg1`, is composed of three layers: an outer layer of brick, a central layer of rock wool (for insulation), and an inner layer of plaster. The door is made of pine, and the window is single-paned. The triangular segment, `seg2`, is composed of four layers: an inner pine layer, a central layer of rock wool, and two outer layers, pine over brick. The `wall` object itself is just an aggregate of the two segments.

Figure 15.24 shows three messages sent to the wall `w`, and the values returned; the last query returns the heat flow through the wall for a temperature difference of 20 degrees.

```
(:area w)                          ⇒ 13
(:opening-area w)                  ⇒ 2.5
(:heat-flow w 20)                  ⇒ 1712.5539190093l
```

Figure 15.24: Messages sent to the sample wall.

We can then change the window in wall `w` to be double-paned, to see what the saving in heat flow will be, if any. This is shown in Figure 15.25. We find that a double-paned window can greatly improve the

overall thermal insulation of the wall.

```
(wOw: (?segl w) (new (Wdw,2Pane,5-4-5 Rectangle) ()
                     (width 1) (height 0.5)))

(:heat-flow w 20)                          => 213.064798857939
```

Figure 15.25: Altering the sample wall.

15.5.5 Observations

This example demonstrates the conciseness with which relatively complex models can be created. The entire wall model as defined in this Section consists of less than 200 lines of code, and some of the model prototypes (i.e. `Material`, `Shape`) could easily be re-used in many other applications.

Furthermore, because design intention is modeled in a relatively straight-forward manner, we conclude that analysis of design entity models created with DESIGNER can accurately reflect on the adequacy of our conceptual models of those entities. For example, both segments of the wall created in Figure 15.23 inherit multiply from `Segment` and from descendants of `Shape` (i.e. `Rectangle` and `Triangle`). It is at this point that the geometric and compositional aspects of the model are combined. These aspects are, insofar as we have defined them here, independent. Their combination has been deferred to the point where it was absolutely necessary. We may have combined `Shape` and `Segment` objects earlier in the development of the model, but this would have led to an increased number of prototypes (i.e. there would have been prototypes for rectangular openings, rectangular segments, triangular openings, and triangular segments). This approach would have introduced a great deal of redundant information that would have made our model more difficult to comprehend. As well, the subsequent addition of other kinds of `Shape` objects (e.g. `Circle`) would have required the addition of circular opening and segment prototypes to maintain consistency with the rest of the model. But as we have done it here, we would only need to define the `Circle` prototype and use it in the creation of various wall segments as required. Thus, an analysis of the computational model of a wall in DESIGNER corresponds to an analysis of the conceptual model underlying the computational one. Such analysis can improve our collective ability to perform design.

Chapter 16

Discussion

DESIGNER represents a new computational paradigm for engineering applications combining the advantages of functional and object-oriented programming paradigms in a seamless and usable system. The functional paradigm lets us use robust formalisms that ensure logical rigor of the resulting system, while object-orientation gives us the ability to model complex entities and relationships directly. Semantic data modeling provides a unique viewpoint on the nature of attributes.

DESIGNER largely satisfies IIM. Since IIM minimizes impedance mismatches with respect to the user's cognitive model of design information by means of its isomorphism (see Part III), DESIGNER also minimizes impedance mismatches with respect to the user's cognitive model.

DESIGNER is not intended to manipulate data, satisfy constraints, perform analysis of design models, or database management. It is a static data modeling language. However, because of Scheme's metacircularity and its ability to operate with higher order functions, it is possible to extend DESIGNER to include dynamic data modeling capabilities (i.e. the capability to operate on and otherwise manipulate data as opposed to its specification).

Experimentation with DESIGNER has included three major examples to date. Firstly, a class-like object has been successfully implemented using DESIGNER's low-level prototyping facilities. This implementation captures all the basic properties of object classes as they are conventionally defined in languages such as SMALLTALK-80. This indicates that our computational model is as expressive as models used by

conventional object-oriented languages, yet simpler and more general than those models.

The second example is the four-bar linkage model. This example demonstrates DESIGNER's ability to capture quite arbitrary design information, and that it is not restricted to modeling the design artifact alone, but can include the artifact's functional specifications and design intent (e.g. the relationships between the specification of the design problem and its solution). The ability of DESIGNER to assist in modeling design artifacts in the conceptual stages of a design process are also indicated.

Third, the example of structural and thermal modeling of walls demonstrates the abilities of DESIGNER and its underlying formal models to capture detailed technical information as well as more general conceptual information about designs. Also, the correspondence between DESIGNER computational models and users' conceptual models facilitates the analysis of designs in general.

The current implementation of DESIGNER is quite compact. The object-oriented extensions to Scheme that form the core of DESIGNER amount to about 300 lines of code. The DESIGNER prototype library is only about 700 lines of code and includes circular lists, queues and stacks, complex numbers, 3D points and vectors, homogeneous 3D transformation matrices, coordinate frames, and some simple parametric solid primitives. DESIGNER currently has no graphics capabilities; but even so, the author feels it is a strong demonstration of the conciseness that can be achieved using object-oriented concepts in a formal functional framework.

DESIGNER currently satisfies all the axioms of HM but two: views and generalization. DESIGNER does permit the creation of sub-objects (subsets), but does not capture the view relationship between an object and its sub-objects explicitly. Views are difficult to deal with because object constraints introduce coupling between attributes in an object. At this time, it appears that each object constraint implicitly defines a view uncoupled from other views; however, it is not clear that providing support only for such uncoupled views is sufficient. Suitable theory will have to be generated regarding the interaction between object constraints and views before the latter can be supported fully by DESIGNER. We note here that DESIGNER has fulfilled its role as a testbed for HM by providing us with this insight regarding its adequacy as a model of design information; however, it does not affect the logical validity of HM, which is preserved.

Generalization (the inverse of specialization) can simplify (normalize) a hierarchy of objects. This simplification can bring to light relationships between objects – and hence between the design entities they model – that may have been obscured by the complexity of the initial hierarchy. Normalization

can also improve the efficiency of operations on the software model. However, there do exist some arguments [28,93] that strongly suggest normalization is of limited usefulness in engineering software systems. These arguments are based on the observation that normalization requires a stable schema (organization) of information to operate correctly. But schema definitions in design tend to exhibit a highly dynamic nature; normalizing a dynamic schema leads to unpredictable and possibly disastrous results. Additionally, generalization implies some fairly complex computation and certainly more so than specialization. For these reasons, the author has elected to defer addition of generalization in DESIGNER to future work.

One final aspect of HM that is not directly supported by the current version of DESIGNER is that of dimensions of measurement, as defined in Section 9.5. Very little work has been done to support dimensions of measurement in computational environments; the author is only aware of the work of Cunis [217]. Because of the lack of available information on the treatment of dimensions of measurement in computational environments upon which to build this capability into DESIGNER, the author has elected to defer the matter to future work.

The distinction between extensional attributes and intentional attributes permits the capture of various kinds of relationships (i.e. constraints) between attributes in a straight-forward manner; as well, this approach is integrated seamlessly into the functional paradigm, thus greatly simplifying the overall computational model. However, this does indicate a further relationship between the modeling of attributes and of constraints. Constraints relate entities at a given degree of abstraction, but logic dictates that the constraints themselves must exist at a higher degree of abstraction. This essentially defines a rule, or criterion, that governs the formation of constraint hierarchies. As was discussed in Section 14.7, the modeling of attributes as intentional or extensional depends on the requirements of the design model; this, in turn, will affect the constraint hierarchy of the model. More research is needed in the area of attribute modeling in order to define the relationship between attributes and constraints more clearly. This constitutes a future extension of HM which the author intends to undertake. This is another insight provided by DESIGNER regarding HM.

DESIGNER employs a canonical message passing mechanism. Though unconventional, the author considers there to be a significant advantage to this approach: the clean separation between function (using generic functions) and structure (using objects) provides a simple, intuitive computational model for

simulation of design artifacts and systems. Furthermore, the notion of *self*, and all the complications that arise from it, are avoided entirely. Although the evaluation of generic functions in the current implementation of DESIGNER is not particularly efficient, several techniques exist that can significantly improve its performance.

In a multi-user environment such as a design group, generic functions offer another potential advantage. Individual group members may *locally* overload particular functions (because, for example, they are so often used by the group member) without affecting the objects and other data structures to which other group members would have access. This would significantly decrease the chances of accidental data corruption.

Finally, the distinct separation of functions that act on objects (i.e. methods in classical object models) from the objects themselves permits the bundling together of groups of functions into modules providing a coarser form of functionality. These modules can be loaded automatically as required, and automatically freed when they are no longer needed, without affecting the data defining the design models themselves. For example, a single design model could be used in two different tasks (e.g. solid modeling and numerical analysis) by simply loading modules of generic functions that provide the functionality needed for each task (e.g. color graphical rendering for solid modeling, versus automatic mesh generation for numerical analysis).

Even though inheritance is generally used in class-based systems, the author has found it to be be usable in a prototype-based system like DESIGNER, where, in conjunction with cloning, it has replaced both subclassing and class instantiation. Furthermore, the use of canonical message-passing eliminates the need for the user to be aware of occurrences of inheritance, and rather treat the relationship as the more intuitive notion of specialization.

DESIGNER permits multiple inheritance only from parent objects that have sets of attributes disjoint from one another. Although this restriction does not exist in other object-oriented languages, the author is constrained to impose it because of the validity requirements of HM. Although the goal of maintaining validity is a desirable one, we must also ask ourselves if this kind of multiple inheritance is enough to satisfy the general requirements of engineering design. There is no simple answer to this question yet; indeed, there is some evidence to suggest that multiple inheritance *per se* is not needed at all, and that the abstractions it provides can be supplied by single inheritance combined with various forms of

aggregation [218]. In the meantime, the author conjectures that multiple inheritance of the kind defined by HM and supported by DESIGNER is sufficient. This is based on the grounds that a counter-example has yet to be found. Obviously, this issue requires further study.

The formal denotational semantics of DESIGNER has yet to be defined. However, Scheme itself is a formalized computer language with a complete denotational semantics [183,184]. Denotational semantics is derived from set theory and the predicate calculus by way of the λ-calculus. Since DESIGNER does not alter Scheme but only extends it within its own formal framework, there *is* a continuity of logical rigor from the formal domain model (HM) through to the actual implementation of DESIGNER. Nonetheless, formalizing DESIGNER's extensions is a worthwhile goal to pursue in order to (a) further corroborate the validity of HM, (b) identify areas where the implementation of DESIGNER may be improved, (c) provide formal proof that DESIGNER in fact satisfies HM (from the point of view of computer science), and (d) provide formal tools to analyze design models generated with DESIGNER.

There exist other various formal semantics (e.g. [208,219]) that define objects in functional programming environments. Some aspects of these efforts are similar to the approaches taken with DESIGNER. The author is therefore confident that a formal semantics for DESIGNER is possible without changes to Scheme's essential structure.

Although the experience with Scheme has indicated to the author that it is a very useful language for such projects as DESIGNER, some difficulties remain: Scheme's syntactic forms can be rather clumsy; support is lacking for certain useful mathematical constructs (such as matrices); the language itself is not tuned to compute efficiently given the important mechanisms DESIGNER requires (i.e. generic functions and object encapsulation). Recent advances in programming language design, however, indicate that these problems may be solved adequately in the near term.

There are many other directions in which DESIGNER can expand in the future. We mention some of them here to indicate the potential for growth.

An argument has been made in [181] that functional programming can permit various degrees of parallelism in computation, and that due to the nature of engineering computing only certain kinds of coarse-grained parallelism can be expected to enhance performance. Elimination of side-effects and strict explicit control of program state vastly simplify parallelization of computation. Objects providing strict encapsulation of information, as in DESIGNER, meet this requirement and may provide the means by which granularity of

parallelism can be made more coarse. The inclusion of an object system in a functional language may thus be an ideal solution. Although DESIGNER does not currently support any parallelism, the author intends to investigate this possibility in the future.

Another area where DESIGNER may find application is in system simulation. Advanced computer modeling and simulation of products has been gaining popularity [220] because of its potential to save a significant amount of time by eliminating the need for physical prototypes. The functional paradigm permits the straight-forward development of quite complex procedural units. Functions to measure time and generate signals are included in many implementations of Scheme. Detailed models of components and design artifacts are also possible in DESIGNER, owing to the object-oriented mechanisms it embodies. Thus, system simulation is also possible *within the same computational model* as are other kinds of engineering computing.

Since Scheme can be used for symbolic manipulation, a system to symbolically manipulate mathematical expressions could be integrated with DESIGNER to provide extensive mathematical support of various kinds of analysis and synthesis in a seamless and integrated way. For example, algebraic functions can be overloaded to operate on symbolic expressions (in Scheme – and hence in DESIGNER as well – symbols such as "x" are acceptable data values that may be operated on). Furthermore, functions that manipulate symbolic representations of equations may be overloaded to provide numerical approximations if numeric data is provided. This would be particularly useful for constraint management [130].

The culmination of the research effort that the author has started and described herein will be a new computer language and associated computational model specifically geared to engineering design. The language and computational model would satisfy HM and be formally defined using denotational (or some other) semantics. This language will provide a computing environment for engineering that will be usable not only for conventional engineering computation, but also as a vehicle for the continued formal study of engineering design.

We note that this approach is itself a strictly formal approach. It makes use of existing, proven tools of computer science and logic, rather than the more hap-hazard means by which many languages currently in use in engineering environments were developed.

Part V

CONCLUSION

Chapter 17

Final Discussion

The contribution of this thesis has been to explain the nature of engineering design information in objective, formal terms.

The formalization of design information must be treated independently of design processes which affect or otherwise manipulate the information. Because of its independence from design processes, such a formalization is universally applicable to any stage or aspect of design. The author has achieved this goal with the Hybrid Model of design information, introduced in Part III. It provides isomorphisms that let us view design information objectively, and its structured notation permits us to reason formally about design information. The abstraction mechanisms introduced in Chapter 10 are founded on ontologic considerations of design. They permit various organizational schemes to be developed, maximizing the amount of available *explicit* information; this in turn minimizes the amount of *interpretation* required, thus increasing confidence in the outcome of actions based on that information. Furthermore, by adhering to the rules regarding the extension of classical set theory, HM is proved valid with respect to its logical foundation (set theory); that is, HM is no less valid than is ZF. HM can facilitate continued research into design by providing a universal formal language for the specification of design information. Its use can improve communications between designers, contribute to the development of effective new taxonomies of design entities and processes, and lead to the creation of more powerful computerized designers' aides.

The use of formal systems greatly clarifies our understanding of design information by helping to resolve difficulties arising from incomplete and vaguely defined nomenclature, managing the changes resulting

from an evolving notion of design, and eliminating sources of logical inconsistencies such as self-reference. Insights of logic have lead the author to introduce conceptual tools – the layered structure of design (Section 6.1), and the notions of a *design space* (Section 6.2) and *artificial science* (Section 5.2) – to help organize our collective design and design research efforts. The results will assist in unifying otherwise incompatible techniques and methodologies by providing a robust and valid reference.

In clearly separating the structure of design itself from the manner in which it is conducted (the *what* of design, versus the *how* of it), the author acknowledges the important role of the designer as the singular agent by which design is manifested. In this sense, logic is seen as the means by which the designer's creativity and intuition can be *channeled* in directions most likely to result in successful solutions to design problems, much as it has been in other "scientific" fields, and forcing us to think more clearly by providing a system wherein logical errors are more easily detected without restricting our freedom to express consistent, relevant information.

In order to demonstrate the advantage of formal systems in design, HM is applied to the development of a new programming language for design (see Part IV). The DESIGNER language is meant to capture arbitrary design information in a flexible framework while largely satisfying HM. In so doing, we provide a bridge between design theory on the one hand and the development of practical computational tools to aid the designer on the other. This continuity of formal rigor has not been achieved before, and increases our confidence in the validity of the language. The unique approaches taken in DESIGNER (e.g. the use of prototypes rather than classes, and canonical rather than conventional message passing) are necessary to meet the requirements of design as a unique and unconventional information management domain. The brevity of its implementation in Scheme is suggestive of the clarity and elegance possible through the use of formal theories to guide the development of engineering software.

Chapter 18

Future Directions

Future research directions for both HM and DESIGNER are discussed in detail at the ends of parts III and IV. Here, the author outlines in a more general sense what the future may hold for the work presented herein.

The key relationship between HM as a formal system and design theory is the isomorphism the author has identified between set theory and design information. The advantage of a formal system with a precise notation is that it lets one see relationships between information in a far simpler and objective way than is possible with more verbose systems requiring significant interpretation (e.g. using the English language instead of mathematical logic). Further study of the axioms of HM will likely bring to light new relationships between kinds of engineering information, and thus improve our understanding of design on the whole. The discovery of new relationships may necessitate modifications or other extensions to HM; this feedback will be beneficial to the development of both HM and design in general. The particular issues that will require further investigation in the near term are aggregations (the primary means by which specific simple design entities are grouped into more complex entities) and constraints (the driving force behind the design process).

Besides the implementation of programming languages for design, HM could be applied to a number of other areas of computer-aided engineering.

The axiomatic form of HM makes it quite amenable to implementation with existing logic programming

languages (e.g. Prolog). The resulting "expert system" would not depend on heuristic knowledge, and would be useful as an analysis tool for design models generated using other computerized tools (e.g. DESIGNER). The use of heuristic knowledge, as has been indicated in the literature survey (Section 2) can markedly reduce the confidence level of expert systems by capturing "knowledge" the soundness of which cannot be proved. Since the number of axioms in HM is quite small, an expert system implementing it may be able to perform in a timely manner.

Another area where HM could find use is in the generation of engineering *databases*. Many operations for object-oriented databases may be written in terms of the axiom of separation of set theory [193]. This implies a relationship between database theory and set theory (and hence HM) which appears promising. There is a well-established precedent in the literature for the general usefulness of object-oriented databases in engineering [56, 102, 106, 168, 221, 222].

One important difference between programming language design and database design is the notion of equality. Currently, in both HM and DESIGNER, identity is defined in terms of structure and behavior of objects; this means that identical objects are permitted to exist. However, in database theory, such identical entities are generally disallowed [223–225] to maintain the database in a normalized form; in other words, the conventional definition of identity in database theory differs from that we have accepted in this work. This discrepancy will have to be addressed before HM can be used to generate useful engineering databases.

Finally, and perhaps most importantly, HM can be used to help teach design in an objective and rigorous manner. It is a relatively small formal system that allows precise definition of terms, and a consistent system for the organization and presentation of information. The actual language of symbolic logic, as used in Part III of this document, need not be introduced immediately: key notions of any theory can be taught by example, as has been done by others [31, 86]. However, the author feels that an introduction to formal systems and symbolic logic sufficient to understand HM would not be a lengthy undertaking and would prove a worthwhile pedagogic investment.

Chapter 19

Closing Remarks

What is engineering design?

This document began with that very question. It is useful now to return to it and determine what headway, if any, has been made towards answering it. Has the author's work presented in this document answered this question? The answer is: to a certain extent, yes.

It may be argued that the question itself is not a particularly good one. It is too vague, too open to interpretation, and tends to invite oversimplified responses. Therefore, some provision is needed to deal with the inherent ambiguity. In keeping with the general tone of this work, the author regards the intention of the question *"What is engineering design?"* as being of a descriptive nature. That is, we do not wish to confuse this question with its procedural complement *"How is engineering design performed?"*

Many of the parameters by which designs are judged are based on quantitative measures. This indicates that design is rooted in the physical world. A designer's "artistic" abilities are constrained insofar as the results of his efforts must comply with the exigencies of the physical world. Furthermore, upon reflection on the arguments made in this document, it is clear that any externalization of the design process may be subjected to logical analysis. Therefore, the author concludes *design must be largely rational in nature.*

This is not a statement of fact, since the rationality of design is not directly observable; nor is the author's work to be construed as a (technical) *proof* of this notion. But the body of evidence herein — the arguments made in Part II, the definition of HM, and its use to create the DESIGNER programming

language — strongly indicate that design has a definite structure that can be treated formally, objectively, and rationally.

We have intentionally said nothing about *how* design is performed. This issue includes the role of the human designer and is prone to effects (such as self-reference) that severely limit the applicability of formal techniques. Nonetheless, some very important headway has been made in this work, headway that represents a strong first step towards a better understanding of engineering design. Although it has traversed the spectrum from the theoretic to the practical, the central theme of this work has been to demonstrate that the use of logic can give us useful and relevant insights into the nature of design. The most important of all, in the opinion of this author, is that logic has been found to be sufficient to explain part of the problem: HM has given us the means of defining the nature of design information in formal and objective terms. It provides the means of capturing, analyzing and communicating design information effectively, efficiently and in a timely manner. It gives us, designers and design researchers, a framework to guide our thinking and our work, by providing a logical system to verify our ideas and thoughts. Upon this foundation new theories of the design process may be built, strengthening our understanding and improving our abilities to meet the challenges of the future.

Part VI

APPENDICES

Appendix A

Source Listings of Designer

A.1 Designer Source

This section lists the source of the Designer language.

```
;;; File        : core.scm
;;; Description : Core Designer functions.
;;; Version     : 1
;;; Revised     : 19-01-93
;;; Copyright   : 1993 by Filippo A. Salustri
;;; Notes       :

(require "bench.o)

;;; MISCELLANEOUS

(define-macro (warn fmt . lst)
  `(print (format #f ,(string-append "Warning: " fmt) ,@lst)))

(define (announce->filename sym)
  (string->symbol (string-append (symbol->string sym) ".scm")))

(define (filename->provide-stmt fn) (list 'provide '',fn))
(define (filename->require-stmt fn) (list 'require '',fn))

(define-macro (announce . l)
  `(begin ,@(map filename->provide-stmt (map announce->filename l))))

(define-macro (needs . l)
  `(begin ,@(map filename->require-stmt (map announce->filename l))))

```

```
(define (flatten-list lst)              ; 'lst' is a list of lists.
  (let loop ((l lst))
    (if (null? l)
        '()
        (append (car l) (loop (cdr l))))))

(define (andmap pred lst)
  (let loop ((l lst))
    (if (null? l)
        #t
        (if (not (pred (car l)))
            #f
            (loop (cdr l))))))

(define (ormap pred lst)
  (let loop ((l lst))
    (if (null? l)
        #f
        (if (pred (car l))
            #t
            (loop (cdr l))))))

(define (filter pred lst)
  (let loop ((l lst))
    (if (null? l)
        '()
        (if (pred (car l))
            (cons (car l) (loop (cdr l)))
            (loop (cdr l))))))

(define not-found 'not-found)
(define (not-found? x) (eq? x not-found))
(define no-val 'no-val)

(define (in-list? val lst)
  (let loop ((l lst))
    (cond
     ((null? l) #f)
     ((eq? val (car l)) #t)
     (else (loop (cdr l))))))

;;; TYPE MODIFICATIONS

(define (attribute? x)
  (and (vector? x)
       (eq? (vector-length x) 3)
       (eq? (vector-ref x 0) 'attribute)))

(define (object? x)
  (and (vector? x)
       (eq? (vector-length x) 4)
       (eq? (vector-ref x 0) 'object)))

(define (generic? x)
  (and (compound? x)
```

```
82          (equal? (procedure-lambda x) the-gf-definition)))
83
84  (define (function? x) (or (procedure? x) (compound? x)))
85
86  ;;; GENERIC FUNCTIONS
87
88  (define (make-signature sig)
89    (vector (if (list? (car sig)) (length (car sig)) -1)
90            (eval `(lambda ,(car sig) (and ,@(cadr sig))))
91            (eval `(lambda ,(car sig) ,@(cddr sig)))))
92
93  (define (eval-signature sig-lst arg-lst)
94    (let loop ((sig-l sig-lst))
95      (if (null? sig-l)
96          not-found
97          (if (and (or (= (vector-ref (car sig-l) 0) -1)
98                       (= (vector-ref (car sig-l) 0) (length arg-lst)))
99                   (apply (vector-ref (car sig-l) 1) arg-lst))
100             (apply (vector-ref (car sig-l) 2) arg-lst)
101             (loop (cdr sig-l))))))
102
103 (define (gf-setter name)
104   (string->symbol (string-append "gfset-" (symbol->string name))))
105
106 (define the-gf-definition '(lambda arg-l
107                              (incr-gfc)
108                              (eval-signature sig-l arg-l)))
109
110 (define-macro (overload name . sig-l)
111   `(begin
112      (if (not (bound? ',name))
113          (begin
114            (incr-gf)
115            (define ,name)
116            (define ,(gf-setter name))
117            (let ((sig-l '()))
118              (set! ,name
119                    (lambda arg-l
120                      (incr-gfc)
121                      (eval-signature sig-l arg-l)))
122              (set! ,(gf-setter name)
123                    (lambda (new-sig-l)
124                      (set! sig-l (append (map make-signature new-sig-l)
125                                          sig-l))
126                      '(((((((          ; return muthin.
127      (,(gf-setter name) ',sig-l))))
128
129 ;;; LOW-LEVEL ATTRIBUTE FUNCTIONS
130
131 (define (Any? value) #t)
132
133 (define (gen-attribute) (vector 'attribute Any? no-val))
134
135 (define (domain a) (vector-ref a 1))
136 (define (set-domain a d) (vector-set! a 1 d))
```

```
137
138  (define (value a) (vector-ref a 2))
139  (define (set-value a v)
140    (if (or (eq? v no-val) ((domain a) v))
141        (vector-set! a 2 v)
142        (error 'value "constraint ~a not satisfied for ~a." (domain a) v)))
143
144  (define (attribute d v)
145    (if (not (procedure? d))
146        (error 'attribute "Domain ~s must be predicates." d)
147    (let ((a (gen-attribute)))
148      (set-domain a d)
149      (set-value a v)
150      a)))
151
152  ;;; THE COPY FUNCTION
153
154  (define (copy thing)
155    (let loop ((x thing))
156      (cond
157       ((attribute? x) (attribute (domain x) (loop (value x))))
158       ((object? x) (clone x))
159       ((pair? x) (cons (loop (car x)) (loop (cdr x))))
160       ((vector? x) (let* ((vl (vector-length x))
161                           (nv (make-vector vl no-val)))
162                      (do ((i 0 (1+ i))) ((>= i vl) nv)
163                        (vector-set! nv i (loop (vector-ref x i))))))
164       (else x))))
165
166  ;;; LOW-LEVEL OBJECT FUNCTIONS
167
168  (define (gen-object)
169    (incr-go)
170    (vector 'object '() '() '()))
171  (define Object (gen-object))            ; the progenitor object.
172
173  (define (slots o) (if (object? o) (vector-ref o 1) '()))
174  (define (add-slot o name v)
175    (vector-set! o 1 (cons (cons name v) (vector-ref o 1))))
176  (define (add-slots o l) (vector-set! o 1 (append l (vector-ref o 1))))
177  (define (set-slots o l) (vector-set! o 1 l))
178
179  (define (parents o) (if (object? o) (vector-ref o 2) '()))
180  (define (add-parent o p) (vector-set! o 2 (cons p (vector-ref o 2))))
181
182  (define (constraints o) (if (object? o) (vector-ref o 3) '()))
183  (define (add-constraint o c) (vector-set! o 3 (cons c (vector-ref o 3))))
184  (define (add-constraints o l) (vector-set! o 3 (append l (vector-ref o 3))))
185
186  (define (: o name . v)
187    (let ((a (assq name (slots o))))
188      (if (not a) (error ': "~a: no such slot name" name))
189      (if v
190          (set-cdr! a (car v))
191          (cdr a))))
```

```
192
193  (define (resolve-object-name x) (if (symbol? x) (eval x) x))
194
195  ;;; OBJECT CONSTRAINTS
196
197  (define (compile-constraint-spec c-spec)
198    (eval `(lambda (obj)
199             (let ,(map (lambda (x) (list x `(value (: obj ',x))))
200                        (car c-spec))
201               ,@(cdr c-spec)))))
202
203  (define-macro (constrain obj . c-spec-l)
204    `(begin (add-constraints ,obj (map compile-constraint-spec ',c-spec-l))
205            '()))
206
207  (define (eval-constraints obj constr-l)
208    (let loop ((c-l constr-l))
209      (if (null? c-l)
210          #t
211          (and ((car c-l) obj)
212               (loop (cdr c-l))))))
213
214  (define (check-object-constraints obj)
215    (eval-constraints obj (constraints obj)))
216
217  ;;; HIGH-LEVEL ATTRIBUTE FUNCTIONS
218
219  (define (attribute-values->list obj)
220    (map value (filter attribute? (map cdr (slots obj)))))
221
222  (define (attribute-names->list obj) (map car (slots obj)))
223
224  ;; for each attribute in obj satisfying pred, map fun.
225  (define (foreach-attribute obj pred fun)
226    (map fun (filter pred (attribute-values->list obj))))
227
228  ;; extensions of 'andmap' and 'ormap' to object attributes.
229  (define (forall obj pred) (andmap pred (attribute-values->list obj)))
230  (define (exists obj pred) (ormap pred (attribute-values->list obj)))
231
232  ;;; HIGH-LEVEL OBJECT FUNCTIONS
233
234  (define (do-inheritance dest src)
235    (add-slots dest (copy (slots src)))
236    (add-parent dest src)
237    (add-constraints dest (copy (constraints src))))
238
239  (define (expand-from-clause obj l)
240    (cond
241     ((list? l)
242      (for-each (lambda (p) (do-inheritance obj p)) (map resolve-object-name l)))
243     (else (do-inheritance obj (resolve-object-name l)))))
244
245  (define (expand-with-clause obj l)
246    (for-each (lambda (spec)
```

```
247                (add-slot obj
248                          (car spec)
249                          (attribute (eval (cadr spec))
250                                     (if (cddr spec)
251                                         (eval (caddr spec))
252                                         no-val))))
253                (let* ((n (car spec))
254                       (q-name (string->symbol
255                                (string-append "?" (symbol->string n))))
256                       (s-name (string->symbol
257                                (string-append (symbol->string n) ":"))))
258                  (if (not (bound? q-name))
259                      (eval `(define (,q-name o) (value (: o ',n)))
260                            top-level-environment))
261                  (if (not (bound? s-name))
262                      (eval `(define (,s-name o v)
263                               (set-value (: o ',n) v)
264                               (if (not (check-object-constraints o))
265                                   (warn "Object constraints not satisfied.")))
266                            top-level-environment))))
267              ))
268
269  (define-macro (new parent-spec with-spec . init-spec)
270    `(let ((new-obj (gen-object)))
271       (expand-from-clause new-obj ',parent-spec)
272       (expand-with-clause new-obj ',with-spec)
273       (if ',init-spec
274           (begin
275             ,@(map (lambda (x)
276                      (list 'set-value `(: new-obj ',(car x)) (cadr x)))
277                    init-spec)))
278       (if (not (check-object-constraints new-obj))
279           (warn "Object constraints not satisfied."))
280       new-obj))
281
282  (define (clone obj)
283    (let ((new-obj (gen-object)))
284      (do-inheritance new-obj obj)
285      new-obj))
286
287  ;;; OTHER OBJECT FUNCTIONS
288
289  (define (parent? child p) (in-list? p (parents child)))
290
291  (define (lineage obj)
292    (if (not (object? obj))
293        '()
294        (cons obj (flatten-list (map lineage (parents obj))))))
295
296  (define (ancestor? obj anc)
297    (if (not (object? obj)) #f
298    (let loop ((o obj))
299      (or (eq? o anc)
300          (ormap (lambda (p) (loop p)) (parents o))))))
301
```

```
302  (define-macro (make-type-predicate o)
303    `(define (,(string->symbol (string-append (symbol->string o) "?")) x)
304       (ancestor? x ,o)))
305
306  (define-macro (make-constructor o . rest)
307    `(define ,(string->symbol (string-append "@" (symbol->string o)))
308       (lambda ,@rest)))
```

A.2 Support/Utility Functions

This section lists various functions used extensively in Designer, but which are not part of Scheme.

```
;;; File        : Preamble.scm
;;; Description : basic functions and primitive overloadings used throughout.
;;; Version     : 4
;;; Revised     : 16-01-93
;;; Copyright   : 1992 by Filippo A. Salustri
;;; Notes       :

; 'orientation-euler' and 'orientation-rpy' from [ref paul].

; Constants ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define PI 3.14159265358979323846)      ; from /usr/include/math.h

; Functions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (number-gt0? x) (and (number? x) (> x 0)))
(define (positive? x) (and (number? x) (>= x 0)))

(define (rad->deg r) (/ (* 180.0 r) PI))
(define (deg->rad d) (/ (* PI d) 180.0))

(define ** expt)

(define (matrix44-x-matrix44 a b)
  (let ((res (vector 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1)))
    (do ((i 0 (+ i 1))) ((= i 4) res)
      (do ((j 0 (+ j 1))) ((= j 4) res)
        (do ((k 0 (+ k 1))
             (v 0 (+ v (* (vector-ref a (+ (* i 4) k))
                          (vector-ref b (+ (* k 4) j))))))
            ((= k 4) (vector-set! res (+ (* i 4) j) v)))))))

(define (vector4-cross a b)
  (vector (- (* (vector-ref a 1) (vector-ref b 2))
             (* (vector-ref b 1) (vector-ref a 2)))
          (- (* (vector-ref a 2) (vector-ref b 0))
             (* (vector-ref b 2) (vector-ref a 0)))
          (- (* (vector-ref a 0) (vector-ref b 1))
             (* (vector-ref b 0) (vector-ref a 1)))
          1))

(define (vector4-dot a b)
  (+ (* (vector-ref a 0) (vector-ref b 0))
     (* (vector-ref a 1) (vector-ref b 1))
     (* (vector-ref a 2) (vector-ref b 2))))

(define (vector4-x-matrix44 v m)
  (let ((res (vector 0 0 0 1)))
    (do ((i 0 (+ i 1))) ((= i 4) res)
```

```
50        (do ((j 0 (+ j 1))
51             (a 0 (+ a (* (vector-ref v j)
52                          (vector-ref m (+ (* j 4) i))))))
53            ((= j 4) (vector-set! res i a))))))
54
55  (define (scalar-x-vector4 s v)
56    (vector (* s (vector-ref v 0))
57            (* s (vector-ref v 1))
58            (* s (vector-ref v 2))
59            1))
60
61  (define (vector4---vector4 a b)
62    (vector (- (vector-ref a 0) (vector-ref b 0))
63            (- (vector-ref a 1) (vector-ref b 1))
64            (- (vector-ref a 2) (vector-ref b 2))
65            1))
66
67  (define (vector4-+-vector4 a b)
68    (vector (+ (vector-ref a 0) (vector-ref b 0))
69            (+ (vector-ref a 1) (vector-ref b 1))
70            (+ (vector-ref a 2) (vector-ref b 2))
71            1))
72
73  (define (vector4-magnitude v)
74    (sqrt (+ (* (vector-ref v 0) (vector-ref v 0))
75             (* (vector-ref v 1) (vector-ref v 1))
76             (* (vector-ref v 2) (vector-ref v 2)))))
77
78  ; Order: rot(z,phi) * rot(y',theta) * rot(z'',psi)
79  ;
80  (define (orientation-euler m)
81    (let* ((phi (if (and (= (vector-ref m 9) 0) (= (vector-ref m 8) 0))
82                    0 (atan (vector-ref m 9) (vector-ref m 8))))
83           (cphi (cos phi))
84           (sphi (sin phi)))
85      (vector phi
86              (atan (+ (* cphi (vector-ref m 8)) ; theta
87                       (* sphi (vector-ref m 9)))
88                    (vector-ref m 10))
89              (atan (- (* cphi (vector-ref m 1)) ; psi
90                       (* sphi (vector-ref m 0)))
91                    (- (* cphi (vector-ref m 5))
92                       (* sphi (vector-ref m 4))))
93              1)))
94
95  ; Order: rot(x,psi) * rot(y,theta) * rot(z,phi)
96  ; with respect to the global (fixed) frame.
97  ;
98  (define (orientation-rpy m)
99    (let* ((phi (if (and (= (vector-ref m 1) 0) (= (vector-ref m 0) 0))
100                   0 (atan (vector-ref m 1) (vector-ref m 0))))
101          (cphi (cos phi))
102          (sphi (sin phi)))
103     (vector (atan (- (* sphi (vector-ref m 8)) ; psi
104                      (* cphi (vector-ref m 9)))
```

```
                    (- (* cphi (vector-ref m 5))
                       (* sphi (vector-ref m 4))))
              (atan (- (vector-ref m 2))   ; theta
                    (+ (* cphi (vector-ref m 0))
                       (* sphi (vector-ref m 1))))
              phi
              1)))

(define (list-last l) (list-ref l (1- (length l))))

(define (list-intersect a b) (filter (lambda (x) (in-list? x b)) a))

; Overloadings ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(overload :show
          ((o) ((object? o)) (format #f "An Object"))
          ((x) () x))
```

Appendix B

Designer Prototype Library

This chapter lists the object prototype definitions available in the Designer library.

B.1 Complex Numbers

```
;;; File        : Complex.scm
;;; Description : complex numbers
;;; Version     : K
;;; Revised     : 14-01-93
;;; Copyright   : 1992 by Filippo A. Salustri
;;; Notes       :

(announce Complex)

;;; Prototype

(make-type-predicate Complex)
(define Complex (new Object
                     ((real number? 0)
                      (imag number? 0))))
(make-constructor Complex (r i) (new Complex () (real r) (imag i)))

;;; Overloadings

(overload :zero? ((c) ((Complex? c))
                      (and (= (?real c) 0) (= (?imag c) 0))))

(overload :magnitude ((c) ((Complex? c))
                          (sqrt (+ (* (?real c) (?real c))
                                   (* (?imag c) (?imag c))))))

```

```
(overload :+
          ((a b) ((Complex? a) (Complex? b))
                 (@Complex (+ (?real a) (?real b)) (+ (?imag a) (?imag b))))
          ((c n) ((Complex? c) (number? n))
                 (@Complex (+ (?real c) n) (?imag c)))
          ((n c) ((Complex? c) (number? n))
                 (@Complex (+ n (?real c)) (?imag c)))))

(overload :-
          ((a b) ((Complex? a) (Complex? b))
                 (@Complex (- (?real a) (?real b)) (- (?imag a) (?imag b))))
          ((c n) ((Complex? c) (number? n))
                 (@Complex (- (?real c) n) (?imag c)))
          ((n c) ((Complex? c) (number? n))
                 (@Complex (- n (?real c)) (?imag c)))))

(overload :*
          ((a b) ((Complex? a) (Complex? b))
                 (@Complex (- (* (?real a) (?real b))
                              (* (?imag a) (?imag b)))
                           (+ (* (?real a) (?imag b))
                              (* (?real b) (?imag a))))))

(overload :/
          ((a b) ((Complex? a) (Complex? b))
                 (if (:zero? b)
                     (error ':/ "Complex divisor is zero.")
                 (@Complex (/ (+ (* (?real a) (?real b))
                                 (* (?imag a) (?imag b)))
                              (+ (* (?real b) (?real b))
                                 (* (?imag b) (?imag b))))
                           (/ (- (* (?real b) (?imag a))
                                 (* (?real a) (?imag b)))
                              (+ (* (?real b) (?real b))
                                 (* (?imag b) (?imag b))))))))

(overload :show
          ((a) ((Complex? a)) (format #f "(~s ~s)" (?real a) (?imag a)))))
```

B.2 3D Spatial Coordinates

```
;;; File        : Coord.scm
;;; Description : 3D coordinates (4vector).
;;; Version     : k
;;; Revised     : 14-01-93
;;; Copyright   : 1992 by Filippo A. Salustri
;;; Notes       :

(announce Coord)
(needs Transform)

;;; Special functions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (vector4->Coord v)
  (@Coord (vector-ref v 0) (vector-ref v 1) (vector-ref v 2)))

; Coord prototype ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(make-type-predicate Coord)
(define Coord (new Object
                   ((x number? 0) (y number? 0) (z number? 0))))
(make-constructor Coord (xv yv zv) (new Coord () (x xv) (y yv) (z zv)))

(overload :show ((c) ((Coord? c))
                    (format #f "(~s ~s ~s)" (?x c) (?y c) (?z c))))

(overload :as-vector ((a) ((Coord? a)) (vector (?x a) (?y a) (?z a) 1)))

(overload :magnitude ((c) ((Coord? c)) (vector4-magnitude (:as-vector c))))

(overload :normalise ((c) ((Coord? c))
                          (let ((m (vector4-magnitude (:as-vector c))))
                            (@Coord (/ (?x c) m) (/ (?y c) m) (/ (?z c) m)))))

(overload := ((a b) ((Coord? a) (Coord? b))
                   (and (= (?x a) (?x b))
                        (= (?y a) (?y b))
                        (= (?z a) (?z b)))))

(overload :cross ((a b) ((Coord? a) (Coord? b))
                        (vector4->Coord
                         (vector4-cross (:as-vector a) (:as-vector b)))))

(overload :dot ((a b) ((Coord? a) (Coord? b))
                      (vector4-dot (:as-vector a) (:as-vector b))))

(overload :*
          ((c t) ((Coord? c) (Transform? t))
                 (vector4->Coord
                  (vector4-x-matrix44 (:as-vector c) (:as-vectors t))))
          ((s c) ((number? s) (Coord? c))
                 (vector4->Coord (scalar-x-vector4 s (:as-vector c))))
          ((c s) ((Coord? c) (number? s))
                 (vector4->Coord (scalar-x-vector4 s (:as-vector c)))))
```

```

(overload :-
          ((a b) ((Coord? a) (Coord? b))
                 (vector4->Coord
                  (vector4---vector4 (:as-vector a) (:as-vector b)))))

(overload :*
          ((a b) ((Coord? a) (Coord? b))
                 (vector4->Coord
                  (vector4-*-vector4 (:as-vector a) (:as-vector b))))

; Subtypes ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define Origin (clone Coord))
```

B.3 Cuboid Parametric Volumes

```
1  ;;; File        : Cuboid.scm
2  ;;; Description : cube-like geometry.
3  ;;; Version     : k
4  ;;; Revised     : 14-01-93
5  ;;; Copyright   : 1992 by Filippo A. Salustri
6  ;;; Notes       :
7
8  (announce Cuboid)
9  (needs Geometry)
10
11 (make-type-predicate Cuboid)
12 (define Cuboid (new Geometry
13                     ((x number-gt0? 1)
14                      (y number-gt0? 1)
15                      (z number-gt0? 1))))
16 (make-constructor Cuboid (xv yv zv) (new Cuboid () (x xv) (y yv) (z zv)))
17
18 (overload :volume ((c) ((Cuboid? c)) (* (?x c) (?y c) (?z c))))
```

B.4 Coordinate Frames

```
;;; File        : Frame.scm
;;; Description : a coordinate frame.
;;; Version     : k
;;; Revised     : 13-01-93
;;; Copyright   : 1992 by Filippo A. Salustri
;;; Notes       :

(announce Frame)
(needs Transform Coord)

(make-type-predicate Frame)
(define Frame (clone Transform))
;;; no constructor yet.

(overload :position ((f) ((Frame? f)) (:* Origin f)))
(overload :orientation ((f) ((Frame? f))
                            (vector4->Coord
                             (orientation-rpy (:as-vectors f)))))

(overload :invert ((f) ((Frame? f))
                       (let ((o (:orientation f))
                             (nf (clone Frame)))
                         (:z-rotate nf (- (?z o)))
                         (:y-rotate nf (- (?y o)))
                         (:x-rotate nf (- (?x o)))
                         nf)))

(overload :translate
          ((f x y z) ((Frame? f))
                     (mat: f (?mat (:* f (:translation x y z))))))
(overload :x-rotate
          ((f r) ((Frame? f)) (mat: f (?mat (:* f (:x-rotation r))))))
(overload :y-rotate
          ((f r) ((Frame? f)) (mat: f (?mat (:* f (:y-rotation r))))))
(overload :z-rotate
          ((f r) ((Frame? f)) (mat: f (?mat (:* f (:z-rotation r))))))
```

B.5 Generalized Geometric Entities

```
;;; File        : Geometry.scm
;;; Description : 3d geometric specs - solid objects.
;;; Version     : k
;;; Revised     : 13-01-93
;;; Copyright   : 1992 by Filippo A. Salustri
;;; Notes       :

(announce Geometry)
(needs Frame)

(make-type-predicate Geometry)
(define Geometry (clone Frame))

(overload :volume ((g) ((Geometry? g))
                       (error "Geometry "Must be implemented in subtypes.")))

(overload :locate
          ((g p wrt) ((Geometry? g) (function? p) (Geometry? wrt))
                     (p (:- (:position g) (:position wrt))
                        (:orientation (:* (:invert wrt) g))))
          ((g p) ((Geometry? g) (function? p))
                 (p (:position g) (:orientation g)))))
```

B.6 3D Lines

```
;;; File        : Line.scm
;;; Description : parametric line segments in 3D
;;; Version     : k
;;; Revised     : 14-01-93
;;; Copyright   : 1992 by Filippo A. Salustri
;;; Notes       :

(announce Line)
(needs Transform Coord)

;;; Line prototype

(make-type-predicate Line)
(define Line (new Object ((start Coord?) (end Coord?))))
(make-constructor Line (a b) (new Line () (start a) (end b)))

(overload := ((a b) ((Line? a) (Line? b))
                    (and (:= (?start a) (?start b))
                         (:= (?end a) (?end b)))))

(overload :* ((l t) ((Line? l) (Transform? t))
                    (@Line (:* (?start l) t) (:* (?end l) t))))

;;; Based on solution in [ref seid] [p 244-245]
;;; There is a degenerate case if 1 line is in the plane formed by the origin
;;; and the other line.  This is dealt with in the 'if'.
(overload :intersect
          ((a b) ((Line? a) (Line? b))
                 (let ((bx (:cross (?start b) (?end b)))
                       (da (:- (?end a) (?start a))))
                   (if (= (:dot bx da) 0)
                       (let ((shift (:translation 0 0 1)))
                         (:* (:intersect (:* a shift) (:* b shift))
                             (:translation 0 0 -1)))
                       (:+ (?start a)
                           (:* (- (/ (:dot bx (?start a))
                                     (:dot bx da)))
                               da))))))

(overload :show
          ((l) ((Line? l))
               (format #f "(~a ~a)" (:show (?start l)) (:show (?end l))))))
```

B.7 Generalized Physical Parts

```
1  ;;; File        : Part.scm
2  ;;; Description : a physical part.
3  ;;; Version     : k
4  ;;; Revised     : 14-01-93
5  ;;; Copyright   : 1992 by Filippo A. Salustri
6  ;;; Notes       :
7
8  (announce Part)
9  (needs Geometry)
10
11 (make-type-predicate Part)
12 (define Part (clone Geometry))
13 ;;; no constructor yet
```

B.8 Queues

```
;;; File        : Queue.scm
;;; Description : FILO lists
;;; Version     : 4
;;; Revised     : 13-01-93
;;; Copyright   : 1992 by Filippo A. Salustri
;;; Notes       :

(announce Queue)
(needs List)

;;; Prototype

(make-type-predicate Queue)
(define Queue (clone List))
(make-constructor Queue arg-l
                  (let ((o (clone Queue)))
                    (for-each (lambda (x) (:push o x)) arg-l)
                    o))

;;; Overloadings

(overload :push ((q v) ((Queue? q)) (:append! q v)))

(overload :pop ((q) ((Queue? q))
                    (if (:empty? q) (error ':pop "nothing left to pop."))
                    (:chop! q)))
```

B.9 Circular Lists

```
;;; File        : Ring.scm
;;; Description : circular list objects.
;;; Version     : k
;;; Revised     : 13-01-93
;;; Copyright   : 1992 by Filippo A. Salustri
;;; Notes       :

(announce Ring)

;;; Prototype

(make-type-predicate Ring)
(define Ring (clone Object))
(add-slot Ring 'lst '())
(add-slot Ring 'offset 0)
(make-constructor Ring l
                  (let ((o (clone Ring)))
                    (: o 'lst l)
                    o))

;;; Overloadings

(overload :first ((r) ((Ring? r))
                      (: r 'offset 0)
                      (list-ref (: r 'lst) 0)))

(overload :length ((r) ((Ring? r)) (length (: r 'lst))))

(overload :as-list ((r) ((Ring? r)) (vector->list (list->vector (: r 'lst)))))

(overload :next ((r) ((Ring? r))
                     (let ((l (length (: r 'lst))))
                       (if (= l 0)
                           (error ':next "zero-sized ring."))
                       (: r 'offset (modulo (1+ (: r 'offset)) l))
                       (list-ref (: r 'lst) (: r 'offset)))))

(overload :peek-next ((r) ((Ring? r))
                          (let ((l (length (: r 'lst))))
                            (if (= l 0)
                                (error ':peek-next "zero-sized ring."))
                            (list-ref (: r 'lst)
                                      (modulo (1+ (: r 'offset)) l)))))

(overload :prev ((r) ((Ring? r))
                     (let ((l (length (: r 'lst))))
                       (if (= l 0)
                           (error ':prev "zero-sized ring."))
                       (: r 'offset (modulo (1- (: r 'offset)) l))
                       (list-ref (: r 'lst) (: r 'offset)))))

(overload :peek-prev ((r) ((Ring? r))
                          (let ((l (length (: r 'lst))))
```

```
54                          (if (= l 0)
55                              (error ':peek-prev "zero-sized ring.")
56                          (list-ref (: r 'lat)
57                                    (modulo (1- (: r 'offset)) l)))))
```

B.10 Stacks

```
;;; File        : Stack.scm
;;; Description : FIFO liste
;;; Version     : 4
;;; Revised     : 13-01-93
;;; Copyright   : 1992 by Filippo A. Salvetti
;;; Notes       :

(announce Stack)
(needs Queue)

;;; Prototype

(make-type-predicate Stack)
(define Stack (clone Queue))
(make-constructor Stack arg-l
                  (let ((o (clone Stack)))
                    (for-each (lambda (x) (:push o x)) arg-l)
                    o))

;;; Overloadings

(overload :push ((s v) ((Stack? s)) (:prepend! v s)))
```

B.11 Geometric Transforms

```
;;; File        : Transform.scm
;;; Description : 3d transforms (4x4 matrices)
;;; Version     : k
;;; Revised     : 14-01-93
;;; Copyright   : 1992 by Filippo A. Salustri
;;; Notes       :

(announce Transform)

;;; Prototype

(make-type-predicate Transform)
(define Transform
  (new Object
       ((mat vector? (vector 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1)))))
;;; no constructor -- probably useless anyways.
;;; maybe could use Loci as arguments in a constructor in the future.

;;; Overloadings

(overload :show ((t) ((Transform? t)) (format #f "~a" (?mat t))))

(overload :as-vectors ((A) ((Transform? A)) (vector-copy (?mat A))))

(overload :* ((A B) ((Transform? A) (Transform? B))
                    (new Transform ()
                         (mat (matrix44-x-matrix44
                               (:as-vectors A) (:as-vectors B))))))

(overload := ((A B) ((Transform? A) (Transform? B))
                    (equal? (?mat A) (?mat B))))

; Subtypes ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define IdentityTransform (clone Transform))

(overload :scaling ((x y z) ((number? x) (number? y) (number? z))
                            (new IdentityTransform ()
                                 (mat (vector x 0 0 0
                                              0 y 0 0
                                              0 0 z 0
                                              0 0 0 1)))))

(overload :translation ((x y z) ((number? x) (number? y) (number? z))
                                (new IdentityTransform ()
                                     (mat (vector 1 0 0 0
                                                  0 1 0 0
                                                  0 0 1 0
                                                  x y z 1)))))

(overload :x-rotation ((x) ((number? x))
                           (new IdentityTransform ()
                                (mat (vector 1 0          0        0
```

```
                                        0 (cos x)     (sin x) 0
                                        0 (- (sin x)) (cos x) 0
                                        0 0           0       1))))))
(overload :y-rotation ((y) ((number? y))
                           (new IdentityTransform ()
                                (mat (vector (cos y) 0 (- (sin y)) 0
                                             0       1 0           0
                                             (sin y) 0 (cos y)     0
                                             0       0 0           1))))))
(overload :z-rotation ((z) ((number? z))
                           (new IdentityTransform ()
                                (mat (vector (cos z)     (sin z) 0 0
                                             (- (sin z)) (cos z) 0 0
                                             0           0       1 0
                                             0           0       0 1))))))
```

Bibliography

[1] Patrick Suppes. *Axiomatic Set Theory*. Dover Publications, Inc., 1972.

[2] S. J. Fenves and W. J. Rasdorf. Treatment of Engineering Design Constraints in a Relational Data Base. *Engineering with Computers*, 1(1):27–37, 1985.

[3] Kenneth J. Waldron. Secret Confessions of A Designer. *Mechanical Engineering*, 114(11):60–62, 1992.

[4] John R. Dixon. The State of Education. *Mechanical Engineering*, pages 64–67, Feb 1991.

[5] R. M. Patel and A. J. McLeod. The Implementation of a Mechanical Engineering Design Interface Using Engineering Features. *Computer-Aided Engineering Journal*, pages 241–246, Dec 1988.

[6] Eugene S. Ferguson. Designing the World We Live In. *Research in Engineering Design*, 4(1):3–11, 1992.

[7] Suresh Konda, Ira Monarch, Philip Sargent, and Eswaran Subrahmanian. Shared Memory in Design: A Unifying Theme for Research and Practice. *Research in Engineering Design*, 4(1):23–42, 1992.

[8] Theodore Bardasz and Ibrahim Zeid. Proposing Analogical Problem Solving to Mechanical Design. In G. L. Kinzel and S. M. Rohde, editors, *Proceedings of the 1990 ASME Computers in Engineering Conference*, pages 181–186. ASME, American Society of Mechanical Engineers, 1990.

[9] M. R. Duffey and J. R. Dixon. A Program of Research in Mechanical Engineering: Computer-Based Models and Representations. MDA Technical Report 11-88, Mechanical Design and Automation Laboratory, University of Massachusetts at Amhert, Amhurst, MA, 1988.

[10] Committee on Engineering Design Theory and Methodology, Manufacturing Studies Board, Commission on Engineering and Technical Systems, and National Research Council. *Improving Engineering Design – Designing for Competitive Advantage*. National Academy Press, 1991.

[11] John R. Dixon. New Goals for Engineering Education. *Mechanical Engineering*, pages 56–62, Mar 1991.

[12] John E. Lockyer. Educating for Awareness: An Alternative to a Five-Year Curriculum. to be presented at the Eigth Canadian Conference on Engineering Education, 1992.

[13] J. R. Dixon, C. D. Jones, E. H. Nielson, S. L. Luby, and E. C. Libardi. Knowledge Representation in Design. MDA Technical Report 1-86, Mechanical Design and Automation Laboratory, University of Massachusetts at Amhert, Amhurst, MA, 1986.

[14] Aart Bijl. An Approach to Design Theory. In H Yoshikawa and E. A. Warman, editors, *Design Theory for CAD*, Proceedings from IFIP WG 5.2 Working Conference on Design Theory for CAD, pages 3–31, Amsterdam, 1987. North-Holland.

[15] Frank G. Pagan. *Formal Specification of Programming Langauges: A Panoramic Primer*. Prentice-Hall, Inc., 1981.

[16] Willard Van Orman Quine. *Mathematical Logic, Revised Edition*. Harvard University Press, 1981.

[17] Zhentao Zhang and Stephen L. Rice. Conceptual Design: Perceiving the Pattern. *Mechanical Engineering*, pages 58–60, Jul 1989.

[18] Thomas W. Malone and John F. Rockart. Computer, Networks and the Corporation. *Scientific American*, 265(3):128–136, Sep 1991.

[19] Robert Fulton. 1991 Shows Increased Attention to Engineering Database Management. in ASME Engineering Database Program Newsletter, Spring, 1992.

[20] Christopher Farrell and Micheal Mandel. Call It What You Will, The Nation Needs A Plan To Nurture Growth. *Business Week*, page 70, Apr 1992.

[21] Thomas G. Dietterich and David G. Ullman. FORLOG: A Logic-based Architecture for Design. In John S. Gero, editor, *Expert Systems in Computer-Aided Design*, Proceedings from IFIP WG 5.2 conference on Expert Systems in Computer-Aided Design, pages 1–17, Amsterdam, 1987. North-Holland.

[22] K. Ishii and P. Barkan. Rule-based Sensitivity Analysis. In John S. Gero, editor, *Expert Systems in Computer-Aided Design*, Proceedings from IFIP WG 5.2 conference on Expert Systems in Computer-Aided Design, pages 179–198, Amsterdam, 1987. North-Holland.

[23] Peter Struss. Multiple Representation of Structure and Function. In John S. Gero, editor, *Expert Systems in Computer-Aided Design*, Proceedings from IFIP WG 5.2 conference on Expert Systems in Computer-Aided Design, pages 57–84, Amsterdam, 1987. North-Holland.

[24] Douglas S. Green and David C. Brown. Qualitative Reasoning During Design about Shape and Fit: a Preliminary Report. In John S. Gero, editor, *Expert Systems in Computer-Aided Design*, Proceedings from IFIP WG 5.2 conference on Expert Systems in Computer-Aided Design, pages 93–112, Amsterdam, 1987. North-Holland.

[25] Gregory L. Fenves. Object-Oriented Programming for Engineering Software Development. *Engineering with Computers*, 6(1):1–15, 1990.

[26] Suad Alagic. *Object-Oriented Database Programming*. Springer-Verlag, New York, 1989.

[27] Stanley B. Zdonik and David Maier. *Fundamentals of Object-Oriented Databases*, pages 1–36. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 1990.

[28] Joan Peckham and Fred Maryanski. Semantic Data Models. *ACM Computing Surveys*, 20(3):153–189, 1988.

[29] J. R. Dixon and M. R. Duffey. The Neglect of Engineering Design. *California Management Review*, 32(2):1–19, 1990.

[30] D. L. Hawla and H. Neishlos. Knowledge Acquisition for Effective and Efficient Use of Engineering Software. *Engineering with Computers*, 6(2):67–80, 1990.

[31] Nam P. Suh. *The Principles of Design*. Oxford University Press, 1990.

[32] Stephen R. Bradley and Alice M. Agogino. Design Capture and Information Management for Concurrent Design. *International Journal of Systems Automation: Research and Applications*, 1(2):117–141, 1991.

[33] J. R. Umaretiya, S. P. Joshi, and S. B. Joshi. An Intelligent Specifications Extraction Interface for Structural Design. *Engineering with Computers*, 6(3):153–165, 1990.

[34] George N. Sandor and Arthur G. Erdman. *Advanced Mechanism Design: Analysis and Synthesis*. Prentice-Hall, Inc., 1984.

[35] M. F. Orelup and J. R. Dixon. Computer-Based Models of Mechanical Design Processes: A Summary of Current Research. MDA Technical Report 9-88, Mechanical Design and Automation Laboratory, University of Massachusetts at Amhert, Amhurst, MA, 1988.

[36] John R. Dixon. Why We Need Doctoral Programs in Design. *Mechanical Engineering*, pages 75–79, Feb 1992.

[37] J. R. Dixon, M. R. Duffey, R. K. Irani, K. L. Meunier, and M. F. Orelup. A Proposed Taxonomy of Mechanical Design Problems. In V. A. Tipnis and E. M. Patton, editors, *Proceedings of the 1988 ASME Computers in Engineering Conference*, pages 41–46. ASME, American Society of Mechanical Engineers, 1988.

[38] John R. Dixon. On Research Methodology Towards a Scientific Theory of Engineering Design. MDA Technical Report 8-88, Mechanical Design and Automation Laboratory, University of Massachusetts at Amhert, Amhurst, MA, 1988.

[39] Tetsuo Tomiyama and Hiroyuki Yoshikawa. Extended General Design Theory. In H. Yoshikawa and E. A. Warman, editors, *Design Theory for CAD*, Proceedings from IFIP WG 5.2 Working Conference on Design Theory for CAD, pages 95–130, Amsterdam, 1987. North-Holland.

[40] Allen C. Ward. A Recursive Model for Managing the Design Process. In James R. Rinderle and David G. Ullman, editors, *Design Theory and Methodology*, ASME Design Automation Conferences, pages 47–52, New York, 1990. ASME.

[41] Owen R. Fauvel. Expanded Use of Function Language in Mechanical Design. In N. Popplewell and A. H. Shah, editors, *Proceedings of the Thirteenth Canadian Congress of Applied Mechanics*, pages 692–693, Winnipeg, Man., 1991. Print Management, Ltd.

[42] J. J. Cunningham and J. R. Dixon. Designing With Features: The Origin of Features. MDA Technical Report 3-88, Mechanical Design and Automation Laboratory, University of Massachusetts at Amhert, Amhurst, MA, 1988.

[43] J. R. Dixon, E. C. Libardi, and E. H. Nielsen. Unresolved Research Issues in Development of Design-With-Features Systems. MDA Technical Report 2-89, Mechanical Design and Automation Laboratory, University of Massachusetts at Amhert, Amhurst, MA, 1989.

[44] J. R. Dixon, J. J. Cunningham, and M. K. Simmons. Research in Designing with Features. MDA Technical Report 4-87, Mechanical Design and Automation Laboratory, University of Massachusetts at Amhert, Amhurst, MA, 1987.

[45] John R. Dixon. Designing with Features: Building Manufacturing Knowledge into More Intelligent CAD Systems. MDA Technical Report 2-88, Mechanical Design and Automation Laboratory, University of Massachusetts at Amhert, Amhurst, MA, 1988.

[46] Raghu Karinthi and Dana Nau. An Approach to Addressing Geometric Feature Interactions. In G. L. Kinzel and S. M. Rohde, editors, *Proceedings of the 1990 ASME Computers in Engineering Conference*, pages 243–250. ASME, American Society of Mechanical Engineers, 1990.

[47] Kosuke Ishii. Role of Computerized Compatibility Analysis in Simultaneous Engineering. *International Journal of Systems Automation: Research and Applications*, 1(4):325–345, 1991.

[48] Jamal A. Abdalla. Version Management Needs for Structural Engineering Design. *Engineering with Computers*, 7(3):131–143, 1991.

[49] M. K. Zamanian, S. J. Fenves, C. R. Thewalt, and S. Finger. A Feature-Based Approach to Structural Design. *Engineering with Computers*, 7(1):1–9, 1991.

[50] R. E. da Silva, K. L. Wood, and J. J. Beaman. Interacting and Interfeature Relationships in Engineering Design for Manufacture. *International Journal of Systems Automation: Research and Applications*, 1(3):263–286, 1991.

[51] Jami J. Shah and Mary T. Rogers. Functional Requirements and Conceptual Design of the Feature-Based Modelling System. *Computer-Aided Engineering Journal*, pages 9–15, Feb 1988.

[52] Frédéric Giacometti and Tien-Chien Chang. Framework to Model Parts, Assemblies, and Tolerances. *International Journal of Systems Automation: Research and Applications*, 1(2):161–181, 1991.

[53] P. H. Ga, H. A. ElMaraghy, and L. Hamid. FDDL: A Feature Based Design Description Language. In W. H. ElMaraghy, W. P. Seering, and D. G. Ullman, editors, *Design Theory and Methodology*, ASME Design Automation Conferences, pages 53–64, New York, 1989. ASME.

[54] Ming-Tzong Wang, Manjula B. Waldron, and R. Allen Miller. Prototype Integrated Feature-Based Design and Expert Process Planning System for Turned Parts. *International Journal of Systems Automation: Research and Applications*, 1(1):7–32, 1991.

[55] James Bowen and Peter O'Grady. A Technology for Building Life-Cycle Design Advisers. In G. L. Kinzel and S. M. Rohde, editors, *Proceedings of the 1990 ASME Computers in Engineering Conference*, pages 1–7. ASME, American Society of Mechanical Engineers, 1990.

[56] Martin Handwick and Blair R. Downie. On Object-Oriented Databases, Materialized Views, and Concurrent Engineering. In V. Saxena, editor, *Engineering Databases: An Engineering Resource*, pages 93–97. ASME, American Society of Mechanical Engineers, 1991.

[57] G. Edward Barton Jr., Robert C. Berwick, and Eric Sven Ristad. *Computational Complexity and Natural Language*. MIT Press, 1987.

[58] A. M. Agogino and A. S. Almgren. Symbolic Computation in Computer-Aided Optimal Design. In John S. Gero, editor, *Expert Systems in Computer-Aided Design*, Proceedings from IFIP WG 5.2 conference on Expert Systems in Computer-Aided Design, pages 267–284, Amsterdam, 1987. North-Holland.

[59] D. Navinchandra and D. H. Marks. Design Exploration Through Constraint Relaxation. In John S. Gero, editor, *Expert Systems in Computer-Aided Design*, Proceedings from IFIP WG 5.2 conference on Expert Systems in Computer-Aided Design, pages 481–509, Amsterdam, 1987. North-Holland.

[60] David A. Hoeltzel and Wei-Hua Chieng. Systems for Unified Life-Cycle Mechanical Engineering Design: Shared-Tool Architectures Versus Distributed Tool Architectures. *Engineering with Computers*, 6(4):211–222, 1990.

[61] John Rasmussen. Shape Optimization and Computer-Aided Design. *International Journal of Systems Automation: Research and Applications*, 1(1):33–45, 1991.

[62] J. K. Wu, F. N. Choong, K. K. Choi, and E. J. Haug. Data Model for Simulation-Based Design of Mechanical Systems. *International Journal of Systems Automation: Research and Applications*, 1(1):67–87, 1991.

[63] George J. Friedman and Cornelius T. Leondes. Constraint Theory, Part I: Fundamentals. *IEEE Transactions on Systems Science and Cybernetics*, ssc-5(1):48–56, 1969.

[64] George J. Friedman and Cornelius T. Leondes. Constraint Theory, Part II: Model Graphs and Regular Relations. *IEEE Transactions on Systems Science and Cybernetics*, ssc-5(2):132–140, 1969.

[65] George J. Friedman and Cornelius T. Leondes. Constraint Theory, Part III: Inequality and Discrete Relations. *IEEE Transactions on Systems Science and Cybernetics*, ssc-5(3):191–199, 1969.

[66] Panos Y. Papalambros and Douglass J. Wilde. *Principles of Optimal Design: Modeling and Computation*. Cambridge University Press, 1988.

[67] David Serrano. Managing Constraints in Concurrent Design: First Steps. In G. L. Kinzel and S. M. Rohde, editors, *Proceedings of the 1990 ASME Computers in Engineering Conference*, pages 159–164. ASME, American Society of Mechanical Engineers, 1990.

[68] Susan Darling Urban and Lois M. L. Delcambre. Constraint Analysis: A Tool for Explaining the Semantics of Complex Objects. In K. R. Dittrich, editor, *Advances in Object-Oriented Databases (Proceedings of 2nd International Workshop on Object-Oriented Database Systems)*, number 334 in Lecture Notes in Computer Science, pages 156–161, Berlin, 1988. Springer-Verlag.

[69] I. R. Grosse and K. Sahu. Extending the Manufacturability Approach to 3-D Configuration Designs. In G. L. Kinzel and S. M. Rohde, editors, *Proceedings of the 1990 ASME Computers in Engineering Conference*, pages 25–32. ASME, American Society of Mechanical Engineers, 1990.

[70] James R. Rinderle and V. Krishnan. Constraint Reasoning in Concurrent Design. In James R. Rinderle and David G. Ullman, editors, *Design Theory and Methodology*, ASME Design Automation Conferences, pages 53–62, New York, 1990. ASME.

[71] Walid Habib and Allen C. Ward. Proving the Labeled Interval Calculus for Inferences on Catalogs. In James R. Rinderle and David G. Ullman, editors, *Design Theory and Methodology*, ASME Design Automation Conferences, pages 63–68, New York, 1990. ASME.

[72] John D. Watton and James R. Rinderle. Impeoving Mechanical Design Decisions with Alternate Formulations of Constraints. In James R. Rinderle and David G. Ullman, editors, *Design Theory and Methodology*, ASME Design Automation Conferences, pages 69–75, New York, 1990. ASME.

[73] Peter J. G. Ramadge and W. Murray Wonham. The Control of Discrete Event Systems. In *Proceedings of the IEEE*, volume 77, pages 81–98. IEEE, 1989.

[74] B. A. Brandin, W. M. Wonham, and B. Benhabib. Manufacturing Cell Supervisory Control - A Timed Discrete Event System. In *International Conference on Robotics and Automation*, pages 531–536. IEEE, 1992.

[75] Martin A. Fogle and J. Kirk Wu. A Relative Coordinate Formulation for Variational Solid Modeling. In James R. Rinderle and David G. Ullman, editors, *Design Theory and Methodology*, ASME Design Automation Conferences, pages 11–18, New York, 1990. ASME.

[76] Ewald Lang, Kai-Uwe Carstensen, and Geoffrey Simmons. *Modelling Spatial Knowledge on a Linguistic Bases*. Number 481 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1991.

[77] Gregory P. Luth, Deepak Jain, Helmut Krawinkler, and Kincho H. Law. A Formal Appeoach to Automating Conceptual Structural Design, Part I: Methodology. *Engineering with Computers*, 7(2):79–89, 1991.

[78] R. H. Crawford and D. C. Anderson. A Computer Representation for Modeling Feedback in Design Processes. In G. L. Kinzel and S. M. Rohde, editors, *Proceedings of the 1990 ASME Computers in Engineering Conference*, pages 165–170. ASME, American Society of Mechanical Engineers, 1990.

[79] Deborah L. Thurston and Tiefu Liu. Design Evaluation of Multiple Attributes under Uncertainty. *International Journal of Systems Automation: Research and Applications*, 1(2):143–159, 1991.

[80] Irving M. Copi. *Symbolic Logic*. Macmillan, 1979.

[81] Kosuke Ishii. The Role of Computers in Simultaneous Engineering. In G. L. Kinzel and S. M. Rohde, editors, *Proceedings of the 1990 ASME Computers in Engineering Conference*, pages 217–224. ASME, American Society of Mechanical Engineers, 1990.

[82] Ravi M. Rangan. Using Information Theory to Model Design Processes Supported by Information Systems. In V. Saxena, editor, *Engineering Databases: An Engineering Resource*, pages 79–86. ASME, American Society of Mechanical Engineers, 1991.

[83] A. J. Medland. *The Computer-Based Design Process*. Springer-Verlag, 1986.

[84] Joseph Constance. DFMA: Learning to Design for Manufacture and Assembly. *Mechanical Engineering*, 114(5):70–74, May 1992.

[85] Steven Ashley. Applying Taguchi's Quality Engineering To Technology Development. *Mechanical Engineering*, 114(7):58–60, July 1992.

[86] David G. Ullman. *The Mechanical Design Process*. McGraw-Hill, 1992.

[87] Wojciech Gasparski. *Understanding Design: The Praxiological-Systemic Perspective*. The Systems Inquiry Series. Intersystems Publications, 1984.

[88] Steven D. Eppinger, Daniel E. Whitney, Robert P. Smith, and David A. Gebala. Organizing the Tasks in Complex Design Projects. In James R. Rinderle and David G. Ullman, editors, *Design Theory and Methodology*, ASME Design Automation Conferences, pages 39–46, New York, 1990. ASME.

[89] R. M. Patel and A. J. McLeod. Engineering Feature Description in Mechanical Engineering Design. *Computer-Aided Engineering Journal*, pages 180–183, Oct 1988.

[90] Srikanth M. Kannapan and Kurt M. Marshek. Design Synthetic Reasoning: A Methodology for Mechanical Design. *Research in Engineering Design*, 2(4):221–238, 1991.

[91] Richard Sause and Graham H. Powell. A Design Process Model for Computer Integrated Structural Engineering. *Engineering with Computers*, 6(3):129–143, 1990.

[92] William H. Slautterback. The Manufacturing Environment in the Year 2000. In *Autofact 6 Conference Proceedings*, pages 21:1–21:9, Michigan, U.S.A, 1984.

[93] Charles M. Eastman. The Contribution of Data Modeling to the Future Development of CAD/CAM Databases. In V. Saxena, editor, *Engineering Databases: An Engineering Resource*, pages 49–54. ASME, American Society of Mechanical Engineers, 1991.

[94] Charles M. Eastman, Alan H. Bond, and Scott C. Chase. A Formal Approach for Product Model Information. *Research in Engineering Design*, 2:65–80, 1991.

[95] Charles M. Eastman. A Data Model Analysis of Modularity and Extensibility in Building Databases. *Building and Environment*, 27(2):135–148, 1992.

[96] C. M. Eastman, A. H. Bond, and S. C. Chase. A Data Model for Design Databases. In J. S. Gero, editor, *Proceedings of Artificial Intelligence in Design '91*, pages 339–365. Butterworth Heinemann, 1991.

[97] Charles M. Eastman, Alan H. Bond, and Scott C. Chase. Application and Evaluation of an Engineering Data Model. *Research in Engineering Design*, 2:185–207, 1991.

[98] Alan H. Bond, Charles M. Eastman, and Scott C. Chase. Theoretical Foundations of EDM Product Design Models. working paper, 1992.

[99] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice Hall International Series in Computer Science. Prentice Hall Inc., 1990.

[100] Saiyid Z. Kamal, H. M. Karandikar, Farrokh Mistree, and Douglas Muster. Knowledge Representation for Discipline-Independent Decision Making. In John S. Gero, editor, *Expert Systems in Computer-Aided Design*, Proceedings from IFIP WG 5.2 conference on Expert Systems in Computer-Aided Design, pages 289–318, Amsterdam, 1987. North-Holland.

[101] R. H. Crawford and D. C. Anderson. Modular Environment for Integrating Preliminary Mechanical Design Software. *International Journal of Systems Automation: Research and Applications*, 1(2):183–202, 1991.

[102] Tetsuo Tomiyama and Paul J. W. Ten Hagen. Orgainization of Design Knowledge in an Intelligent CAD Environment. In John S. Gero, editor, *Expert Systems in Computer-Aided Design*, Proceedings from IFIP WG 5.2 conference on Expert Systems in Computer-Aided Design, pages 119–147, Amsterdam, 1987. North-Holland.

[103] B. T. David. Multi-Expert Systems for CAD. In P. J. W. ten Hagen and Tetsuo Tomiyama, editors, *Intelligent CAD Systems I (Theoretical and Methodological Aspects)*, EurographicSeminar Series, pages 57–67, Berlin, 1987. Springer-Verlag.

[104] J. F. Koegel. A Theoretical Model for Intelligent CAD. In P. J. W. ten Hagen and Tetsuo Tomiyama, editors, *Intelligent CAD Systems I (Theoretical and Methodological Aspects)*, EurographicSeminar Series, pages 206–223, Berlin, 1987. Springer-Verlag.

[105] Y. E. Kalay, L. M. Swerdloff, and A. C. Harfmann. A Knowledge-based Computable Model of Design. In John S. Gero, editor, *Expert Systems in Computer-Aided Design*, Proceedings from IFIP WG 5.2 conference on Expert Systems in Computer-Aided Design, pages 203–223, Amsterdam, 1987. North-Holland.

[106] Kincho H. Law, Thierry Barsalou, and Gio Wiederhold. Management of Complex Structural Engineering Objects in a Relational Framework. *Engineering with Computers*, 6(2):81–92, 1990.

[107] Guidong Han, Setsuo Ohsuga, and Hiroyuki Yamauchi. The Application of Knowledge Base Technology to CAD. In John S. Gero, editor, *Expert Systems in Computer-Aided Design*, Proceedings from IFIP WG 5.2 conference on Expert Systems in Computer-Aided Design, pages 25–51, Amsterdam, 1987. North-Holland.

[108] E. F. Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley, 1990.

[109] Ronald G. Ross. *Entity Modeling: Techniques and Application*. Database Research Group, Inc., 1987.

[110] James J. Odell. Modelling Object Using Binary- and Entity-Relationship Models. *Journal of Object-Oriented Programming*, 5(3):12–18, Jun 1992.

[111] Richard Hull and Roger King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[112] Kincho H. Law, Gio Wiederhold, Niki Siambela, Walter Sujansky, David Zingmond, and Harvinder Singh. Architecture for Managing Design Objects in a Shareable Relational Framework. *International Journal of Systems Automation: Research and Applications*, 1(1):47–65, 1991.

[113] David Stemple, Adolfo Socorro, and Tim Sheard. Formalizing Objects for Databases using AD-ABTPL. In K. R. Dittrich, editor, *Advances in Object-Oriented Databases (Proceedings of 2nd International Workshop on Object-Oriented Database Systems)*, number 334 in Lecture Notes in Computer Science, pages 110–128, Berlin, 1988. Springer-Verlag.

[114] David L. Stubbs and P. Derek Emes. Modularization: Prefabricating a Process Plant. *Mechanical Engineering*, pages 63–65, Nov 1990.

[115] no author given. Object Oriented Databases: An Engineer's Perspective. in ASME Engineering Database Program Newsletter, Spring, 1992.

[116] staff. Objective Data. Scientific American, Science and Business Department, Mar 1992.

[117] Johnathan S. Colton and John L. Dascanio, II. An Integrated, Intelligent Design Environment. *Engineering with Computers*, 7(1):11–22, 1991.

[118] Won Kim. *Introduction to Object-Oriented Databases*. The MIT Press, 1990.

[119] Herbert B. Voelcker. Modeling in the Design Process. In W. Dale Compton, editor, *Design and Analysis of Integrated Manufacturing Systems*, pages 167–199, Washington, DC, 1988. National Academy Publishing.

[120] G. Sunde. A CAD System with Declarative Specification of Shape. In P. J. W. ten Hagen and Tetsuo Tomiyama, editors, *Intelligent CAD Systems I (Theoretical and Methodological Aspects)*. EurographicSeminar Series, pages 90–103, Berlin, 1987. Springer-Verlag.

[121] Stuart Watson. Relational Geometry – a New Generation of Two-Dimensional CAD. *Computer-Aided Engineering Journal*, pages 169–172, Aug 1988.

[122] Walid Keirouz, Jahir Pabon, and Robert Young. Integrating Parametric Geometry, Features and Variational Modeling for Conceptual Design. In James R. Rinderle and David G. Ullman, editors, *Design Theory and Methodology*, ASME Design Automation Conferences, pages 690–691, New York, 1990. ASME.

[123] Xin Dong and Micheal Wozny. Managing Feature Type Dependency in a Feature-Based Modeling System. In G. L. Kinzel and S. M. Rohde, editors, *Proceedings of the 1990 ASME Computers in Engineering Conference*, pages 125–130. ASME, American Society of Mechanical Engineers, 1990.

[124] U. Roy and C. R. Liu. Establishment of Functional Relationships between Product Components in Assembly Database. *Computer-Aided Design*, 20:570–580, Dec 1988.

[125] Uwe Weissflog. CIM Dictionaries: An Evolutionary Approach to CIM Data Integration. In V. Saxena, editor, *Engineering Databases: An Engineering Resource*, pages 71–78. ASME, American Society of Mechanical Engineers, 1991.

[126] Ravi M. Rangan and Robert E. Fulton. A Data Management Strategy to Control Design and Manufacturing Information. *Engineering with Computers*, 7(2):63–78, 1991.

[127] Gerald Jay Sussman and Guy Lewis Steele Jr. CONSTRAINTS – A Language for Expressing Almost-Hierarchical Descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[128] Glen Mullineux. A Blackboard Structure for Handling Engineering Design Data. *Engineering with Computers*, 7(3):185–195, 1991.

[129] Philip M. Sargent. Materials Data Interchange for Component Manufacture. *Engineering with Computers*, 6(4):237–247, 1990.

[130] David Serrano. Constraint-Based Concurrent Design. *International Journal of Systems Automation: Research and Applications*, 1(3):287–304, 1991.

[131] Julian Jaynes. *The Origin of Consciousness in the Breakdown of the Bicameral Mind*. University of Toronto Press, 1976.

[132] Yogesh Jaluria and D. Lombardi. Use of Expert Systems in the Design of Thermal Equipment and Processes. *Research in Engineering Design*, 2(4):239–253, 1991.

[133] I. Popescu and M. B. Zaremba. An Efficient Search Method for Expert Robot Control. *International Journal of Systems Automation: Research and Applications*, 1(4):369–386, 1991.

[134] Enn H. Tyugu. Merging Conceptual and Expert Knowledge in CAD. In John S. Gero, editor, *Expert Systems in Computer-Aided Design*, Proceedings from IFIP WG 5.2 conference on Expert Systems in Computer-Aided Design, pages 423–431, Amsterdam, 1987. North-Holland.

[135] John C. Tang and Larry J. Leifer. An Observational Methodology for Studying Group Design Activity. *Research in Engineering Design*, 2(4):209–219, 1991.

[136] M. R. Cutkosky and J. M. Tenenbaum. Providing Computational Support for Concurrent Engineering. *International Journal of Systems Automation: Research and Applications*, 1(3):239–261, 1991.

[137] Avron Barr and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence, Volume 1*. HeurisTech Press, Stanford, California, 1981.

[138] Avron Barr and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence, Volume 2*. HeurisTech Press, Stanford, California, 1982.

[139] Paul R. Cohen and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence, Volume 3*. HeurisTech Press, Stanford, California, 1982.

[140] Steven Ashley. The Battle to Build Better Products. *Mechanical Engineering*, pages 34–38, Nov 1990.

[141] Steven Ashley. DARPA Initiative in Concurrent Engineering. *Mechanical Engineering*, 114(4):54–57, Apr 1992.

[142] Martin Cwiakala. Using Hypermedia Concepts to Enhance CAD. In G. L. Kinzel and S. M. Rohde, editors, *Proceedings of the 1990 ASME Computers in Engineering Conference*, pages 131–137. ASME, American Society of Mechanical Engineers, 1990.

[143] Mark H. Chignell. A Taxonomy of User Interface Terminology. *ACM SIGCHI Bulletin*, 21(4):27–34, 1990.

[144] Christoph Hubel and Bernd Sutter. Supporting Engineering Applications by New Data Base Processing Concepts – An Experience Report. *Engineering with Computers*, 8(1):31–49, 1992.

[145] Katia P. Sycara and C. Micheal Lewis. Modeling Group Decision Making and Negotiation in Concurrent Product Design. *International Journal of Systems Automation: Research and Applications*, 1(3):217–238, 1991.

[146] D. Sriram, R. Logcher, A. Wong, and S. Ahmed. Computer-Aided Cooperative Product Development: A Case Study. *International Journal of Systems Automation: Research and Applications*, 1(1):89–112, 1991.

[147] George J. Klir and Tina A. Folger. *Fuzzy Sets, Uncertainty, and Information*. Prentice-Hall, 1988.

[148] Lotfi A. Zadeh. Knowledge Representation in Fuzzy Logic. *IEEE Transactions on Knowledge and Data Engineering*, 1(1):89–99, Mar 1989.

[149] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Vintage Books, 1979.

[150] Joe Halpern. Reasoning about knowledge. class notes for CS2422S, University of Toronto, 1990.

[151] Richard Sause and Graham H. Powell. A Design Process Model for Computer Integrated Structural Engineering: Design Phases and Tasks. *Engineering with Computers*, 7(3):145–160, 1991.

[152] Clive L. Dym and Raymond E. Levitt. Toward the Integration of Knowledge for Engineering Modeling and Computation. *Engineering with Computers*, 7(4):209–224, 1991.

[153] Abraham A. Fraenkel, Yehoshua Bar-Hillel, and Azriel Levy. *Foundations of Set Theory*. North-Holland, 1973.

[154] Morris Kline. *Mathematics: The Loss of Certainty*. Oxford University Press, 1980.

[155] Gerald D. Fischbach. Mind and Brain. *Scientific American*, 267(3):48–57, Sep 1992.

[156] Geoffrey E. Hinton. How Neural Networks Learn from Experience. *Scientific American*, 267(3):144–151, Sep 1992.

[157] Francis Crick and Christof Koch. The Problem of Consciousness. *Scientific American*, 267(3):152–159, Sep 1992.

[158] John D. Watton and James R. Rinderle. A Method to Identify Reformulations of Mechanical Parametric Constraints to Enhance Design. In James R. Rinderle and David G. Ullman, editors, *Design Theory and Methodology*, ASME Design Automation Conferences, pages 77–84, New York, 1990. ASME.

[159] Bjarne Stroustrup. What is Object-Oriented Programming? *IEEE Software*, pages 10–20, May 1988.

[160] John B. Smith and Stephen F. Weiss. Hypertext. *CACM*, 31(7):816–819, Jul 1988.

[161] J. Wade and J. S. Colton. A Framework for Feature-Based Representation of the Design Process. *Engineering with Computers*, 6(3):185–192, 1990.

[162] Sonia Enand, Dan Small, and Keith Williamson. Developing Core Language Constructs for Knowledge-Based Engineering Systems. *International Journal of Systems Automation: Research and Applications*, 2(4):319–333, 1992.

[163] K. G. Swift. *Knowledge-Based Design for Manufacture*. Prentice-Hall, Inc., 1987.

[164] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *ACM OOPS Messenger*, pages 8–87, Jun 1990.

[165] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[166] Will Durant. *The Story of Philosophy*. Pocket Books, 1961.

[167] Kirk Martini and Graham H. Powell. Geometric Modeling Requirements for Structural Design. *Engineering with Computers*, 6(2):93–102, 1990.

[168] Jintae Lee and Thomas W. Malone. Partially Shared Views: A Scheme for Communicating among Groups that Use Different Type Hierarchies. *ACM Transactions on Information Systems*, 8(1):1–26, Jan 1990.

[169] Steven J. Fenves. personal communication, 192.

[170] John Bell and Moshe Machover. *A Course in Mathematical Logic*. Elsevier Science Publishers, Amsterdam, 1977.

[171] Micheal J. Wozny. Beyond Computer Graphics and CAD/CAM. In Tosiyasu L. Kunii, editor, *Proceedings of Computer Graphics International '87*, pages 3–9, Tokyo, 1987. Springer-Verlag.

[172] Stefan Dessloch, Christoph Hilbel, Nelson Mendonca Mattos, and Bernd Sutter. Handling Functional Constraints of Technical Modeling Systems in a KBMS Environment. *International Journal of Systems Automation: Research and Applications*, 1(4):347–367, 1991.

[173] P. Bernus and Z. Letray. Intelligent Systems Interconnection: What Should Come After Open Systems Interconnection? In P. J. W. ten Hagen and Tetsuo Tomiyama, editors, *Intelligent CAD Systems I (Theoretical and Methodological Aspects)*, EurographicSeminar Series, pages 44–55, Berlin, 1987. Springer-Verlag.

[174] Dragan D. Djakovic. RASP - A Language with Operations on Fuzzy Sets. *Computer Languages*, 13(3):143–147, 1988.

[175] Allen C. Ward. Some Language-Based Approaches to Concurrent Engineering. *International Journal of Systems Automation: Research and Applications*, 2(4):335–351, 1992.

[176] Samuel N. Kamin. *Programming Languages: An Interpreter-Based Approach*. Addison-Wesley, 1990.

[177] Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1990.

[178] Terrence W. Pratt. *Programming Languages: Design and Implementation*. Prentice-Hall, Inc., 1975.

[179] Robert R. Kessler. *LISP, Objects and Symbolic Programming*. Scott, Foresman and Company, 1988.

[180] Anthony J. Field and Peter G. Harrison. *Functional Programming*. International Computer Science Series. Addison-Wesley, 1988.

[181] John W. Baugh Jr. and Daniel R. Rehak. Applications of Coarse-Grained Dataflow in Computational Mechanics. *Engineering with Computers*, 8(1):13–30, 1992.

[182] David A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall International Series in Computer Science. Prentice Hall Inc., 1991.

[183] Revised^3 Report on the Algorithmic Language Scheme. *ACM Sigplan Notices*, 21(12), Dec 1986.

[184] IEEE Standard for the Scheme Programming Language. IEEE Std 1178-1990, 1991. Institute of Electrical and Electronic Engineers.

[185] Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, Inc., 1987.

[186] Oliver Laumann and Carsten Bormann. Elk: The Extension Language Kit. working paper, included with source, 1992.

[187] Guy Argo, John Hughes, Philip Trinder, Jon Fairbairn, and John Launchbury. *Implementing Functional Databases*, chapter 10, pages 165–176. ACM Press Frontier Series. Addison-Wesley, 1990.

[188] Oscar Nierstrasz. *A Survey of Object-Oriented Concepts*, chapter 1, pages 3–21. ACM Press Frontier Series. Addison-Wesley, 1989.

[189] Kenneth Kahn, Eric Dean Tribble, Mark S. Miller, and Daniel G. Bobrow. *Vulcan: Logical Concurrent Objects*, pages 75–112. Computer Systems Series. The MIT Press, 1987.

[190] Gio Wiederhold. Views, Objects, and Databases. *IEEE Computer*, 19(12):37–44, Dec 1986.

[191] H. H. Lee and J. S. Arora. Object-Oriented Programming for Engineering Applications. *Engineering with Computers*, 7(4):225–235, 1991.

[192] Bruce W. R. Forde, Ricardo O. Foschi, and Siegfried F. Stiemer. Object-Oriented Finite Element Analysis. *Computers and Structures*, 34(3):355–374, 1990.

[193] Gail M. Shaw and Stanley B. Zdonik. A Query Algebra for Object-Oriented Databases. Technical Report CS-89-19, Department of Computer Science, Brown University, Providence, Rhode Island, Mar 1989.

[194] Micheal Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. Technical Report 90/14 (2nd revision), Department of Computer Science, SUNY, Stony Brook, NY, Aug 1990.

[195] José Meseguer. A Logical Theory of Concurrent Objects. In Norman Meyrowitz, editor, *Conference on Object-Oriented Programming: Systems, Languages, and Applications*, New York, 1990. ACM Press.

[196] Various authors. X3/SPARC/DBSSG/OODBTG Final Report. Technical report, Accredited Standards Committee X3, 1991.

[197] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[198] Bjarne Stroustrup. A C++ Tutorial. Computer Science Technical Reports 113, AT&T Bell Labs, Murray Hill, NJ, Sep 1984.

[199] Ralph E. Johnson and Jonathan M. Zweig. Delegation in C++. *Journal of Object-Oriented Programming*, 4(7):31–34, 1991.

[200] Various Authors. The SELF Papers. Technical Report CIS 209, Stanfood University, Stanford, CA, 1991.

[201] Wolfgang Kreutzer and Malcolm Stairmand. C-Flavours: A Scheme-Based Flavour System with Coroutines and its Applications to the Design of Object-Oriented Simulation Software. *Computer Languages*, 15(4):225–249, 1990.

[202] Stephen Slade. *The T Programming Language: a dialect of LISP*. Prentice-Hall, Inc., 1987.

[203] Joseph A. Goguen and Jose Meseguer. *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics*, pages 417–477. Computer Systems Series. The MIT Press, 1987.

[204] Christian Queinnec. Meroon: A Small, Efficient and Enhanced Object System. unpublished book chapter, July 1992.

[205] Sho-Huan Simon Tung. *Merging Interactive, Modular and Object-Oriented Programming*. PhD thesis, Department of Computer Science, Indiana University, 1992.

[206] Kamran Parsaye, Mark Chignell, Setrag Khoshafian, and Harry Wong. *Intelligent Databases: Object-Oriented, Deductive Hypermedia Technologies*. John Wiley and Sons, Inc., 1989.

[207] Norman Adams and Jonathan Rees. Object-Oriented Programming in Scheme. In Jerome Chailloux, editor, *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 277–288, New York, 1988. ACM.

[208] Uday S. Reddy. Objects as Closures: Abstract Semantics of Object Oriented Languages. In Jerome Chailloux, editor, *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 289–297, New York, 1988. ACM.

[209] Luca Cardelli. *A Semantics of Multiple Inheritance*, pages 59–83. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 1990.

[210] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[211] Alan Snyder. *Inheritance and the Development of Encapsulated Software Components*, pages 165–188. Computer Systems Series. The MIT Press, 1987.

[212] Ibrahim Zeid. *CAD/CAM: Theory and Practice*. McGraw-Hill, Inc., 1991.

[213] Richard P. Paul. *Robot Manipulators: Mathematics, Programming and Control*. MIT Press, 1981.

[214] William C. Reynolds and Henry C. Perkins. *Engineering Thermodynamics*. McGraw-Hill, 1977.

[215] William H. Beyer, editor. *CRC Standard Mathematical Tables*. CRC Press, Inc., 27 edition, 1984.

[216] Brent Hailpern and Van Nguyen. *A Model for Object-Based Inheritance*, pages 147–164. Computer Systems Series. The MIT Press, 1987.

[217] Roman Cunis. A Package for Handling Units of Measure in Lisp. *ACM LISP Pointers*, 5(2):21–25, 1992.

[218] Edward Sciore. Object Specialization. *ACM Transactions on Information Systems*, 7(2):103–122, Apr 1989.

[219] Shinn-Der Lee and Daniel P. Friedman. First-Class Extents. Technical Report 350, Computer Science Department, Indiana University, Bloomington, Indiana, Mar 1992.

[220] Micheal Puttré. Virtual Prototypes Move Alongside Their Physical Counterparts. *Mechanical Engineering*, pages 59–61, Aug 1992.

[221] H. N. An-Nashif and G. H. Powell. An Object-Oriented Alorithm for Automated Modeling of Frame Structures: Stiffness Modeling. *Engineering with Computers*, 7(2):121–128, 1991.

[222] Hamideh Afsarmanesh. The 3Dis: An Extensible Object-Oriented Information Management Environment. *ACM Transactions on Information Systems*, 7(4):339–377, Oct 1989.

[223] Grady Booch. *Object Oriented Design with Applications*. Benjamin/Cummings Publishing Company, Inc., 1991.

[224] Steve Ford, John Joseph, David E. Langworthy, David F. Lively, Girish Pathak, Edward R. Perez, Robert W. Peterson, Diana M. Sparacin, Satish M. Thatte, David L. Wells, and Sanjive Agarwala. Zeitgeist: Database Support for Object-Oriented Programming. In K. R. Dittrich, editor, *Advances in Object-Oriented Databases (Proceedings of 2nd International Workshop on Object-Oriented Database Systems)*, number 334 in Lecture Notes in Computer Science, pages 23–42, Berlin, 1988. Springer-Verlag.

[225] Setrag Khoshafian and Dan Frank. Implementation Techniques for Object Oriented Databases. In K. R. Dittrich, editor, *Advances in Object-Oriented Databases (Proceedings of 2nd International Workshop on Object-Oriented Database Systems)*, number 334 in Lecture Notes in Computer Science, pages 60–80, Berlin, 1988. Springer-Verlag.