Languages for Knowledge Capture and their Use in Creation of Smart Models

Biren Prasad[1]

[1]Unigraphics Solutions, Knowledge-based Engineering (KBE) PBUs , CERA Institute, P.O. Box 3882, Tustin, CA 92782, Tel: (714) 952-5562, Fax: (714) 505-0663, Email: <prasadb@ugsolutions.com>

Abstract: Languages are means of capturing the knowledge for the design and development of a product. Smart Models are the results of such knowledge capture. The author, first describes how languages for knowledge capture have evolved over a thirty year time period. Author through literature search finds such languages to fall into three major classes: (a) Geometry-based language (b) Constraint-based language (c) Knowledge-based language. The paper then describes the differences and similarities of these languages that can be employed to capture life-cycle intent. The second part of the paper describes how such languages are being used in creation of smart models. A smart model is a reusable conceptualization of an application domain. The models contain the knowledge (attributes, rules or relations) of the application domains forming the basis for future problem solving. The paper also describes two popular ways of formulating a problem that leads to such smart models: (1) Constraint-based programming (2) Knowledge-based programming. Through analysis of existing practices, new development and trends, the paper then discusses some "new emerging directions in the use of languages for the knowledge capture". Finally, the benefits of knowledge capture and creation of smart models over conventional models are discussed.

1. Introduction

Except in a few rare cases, products are now so complex that it is extremely difficult to correctly "capture" their life-cycle intent right the first time no matter what C4 (CAD/CAM/CAE/CIM) tools, productivity gadgets or automation widgets are used. Traditionally, CAD tools are primarily used for activities that occur at the end of the design process. Such usage of CAD tools, for instance, during detailing geometry of an artifact, is in generating a production drawing, or in documenting geometry in a digitized form (See Figure 1). CAM systems are conventionally used to program machining or cutting instructions on the NC machines for a part whose mock-up design, clay or plaster prototype may already exist. CAE systems are used to check the integrity of the designed artifact (such as structural analysis for stress, thermal, etc.), when most of the critical design decisions have already been made.

Studies have revealed that 75% of the eventual cost of a product is determined before any full-scale development or a CAD tool usage actually begins [1].

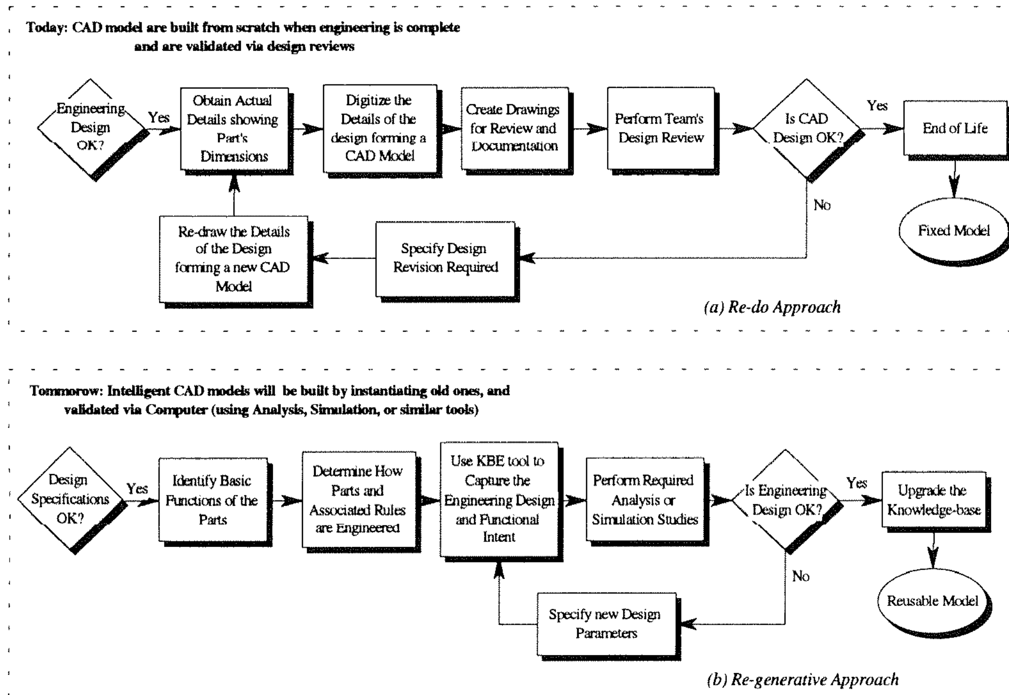


Figure 1: Re-do versus Reusable Approach to CAD Modeling (a) Re-do Approach (b) Re-generative Approach

Most C4 tools in use today are not really "***capture***" tools. The need for "***change***" after a design model is initially "***built***" is all but inevitable. Today, CAD models are built from scratch only when engineering activity is complete, and are validated via a series of design reviews. Design work-groups typically document the design through CAD software only after the completion of major engineering processes and after resolving all of the pressing engineering issues. A work-group captures the geometry in a static form, such as lines and surfaces. Static representation is actually a documentation that tells a designer what the final design looks like but not how it has come to be. If changes are required in the design, a new CAD model is recreated (see Figure 1) using some types of computer-aided "re-do" or "back-tracking" methods. Such CAD methods of activating change or modification (e.g., a redo or a backtracking) can be extremely time-consuming and costly being that late in the life-cycle process. In such static representations of geometry, configuration changes cannot be handled easily, particularly when parts and dimensions are linked. In addition to the actual process that led to the final design, most of the useful lessons learned along the way are also lost. In the absence of the latter, such efforts have resulted in loss of configuration control, proliferation of changes to fix the errors caused by other changes, and sometimes-ambiguous designs. Hence, in recent days, during a PD^3 process, emphasis is often placed on the methods used for capturing the life-cycle intent with ease of modifications in

mind. The power of a "capture" tool comes from the methods used in capturing the design intent initially so that the anticipated changes can be made easily and quickly later if needed. By capturing a "design intent" as opposed to a "static geometry," configuration changes could be made and controlled more effectively using the power of the computer than through the traditional CAD attributes (such as line and surfaces). "*Life-cycle capture*" refers to the definition of a physical object and its environment in some generic form [2]. "*Life-cycle intent*" means representing a life-cycle capture in a form that can be modified and iterated until all the life-cycle specifications for the product are fully satisfied. "*Design-capture*" likewise refers to the design definitions of the physical objects and its surroundings. "*Design-intent*" means representing the "*design capture*" in a form (such as a parametric or a variational scheme) that can be iterated. Design in this case means one of the life-cycle functions (see Figure 4.2 of Volume I [3]). In the future, CAD models will be reusable. The new models will be built by instantiating the old ones and validating them via computer (using simulation, analysis, sensitivity, optimization, etc., see Figure 1). Such models will have some level of intelligence built into them [4].

2. Languages for Life-cycle Capture

Languages are means of capturing the knowledge for the design and development of a product. Models are the results of such knowledge capture. The primary goal of knowledge-capture formalism is to provide a means of defining ontology. Ontology is a set of basic attributes and relations comprising the vocabulary of the product realization domain as well as rules for combining the attributes and relations. Engineering Analysis Language (EAL), for example, provides a means of creating analysis or design models as run streams. Later, they form the basis for iterative analysis and design [5]. ICAD/IDL, on the other hand, captures the knowledge about the process of designing and developing a product [6][7]. There are three types of languages that can be employed to capture life-cycle intent:

a) Geometry-based language
b) Constraint-based language
c) Knowledge-based language

Figure 2 shows an evolution of languages for capturing knowledge over a thirty year time period. These are C4 (CAD/CAM/CIM/CAE) specification languages for product engineers or designers to define configurations of parts and assemblies. They are not computer languages for software programmers (such as C, or C++). During this thirty-year period, there had been a tremendous innovation.

- The ***first generation*** of C4 languages, first introduced during 1960, only dealt with 2-D drafting and 2-D wire-frame design.
- The ***second generation*** of C4 languages dealt with surfaces and 3-D solids.

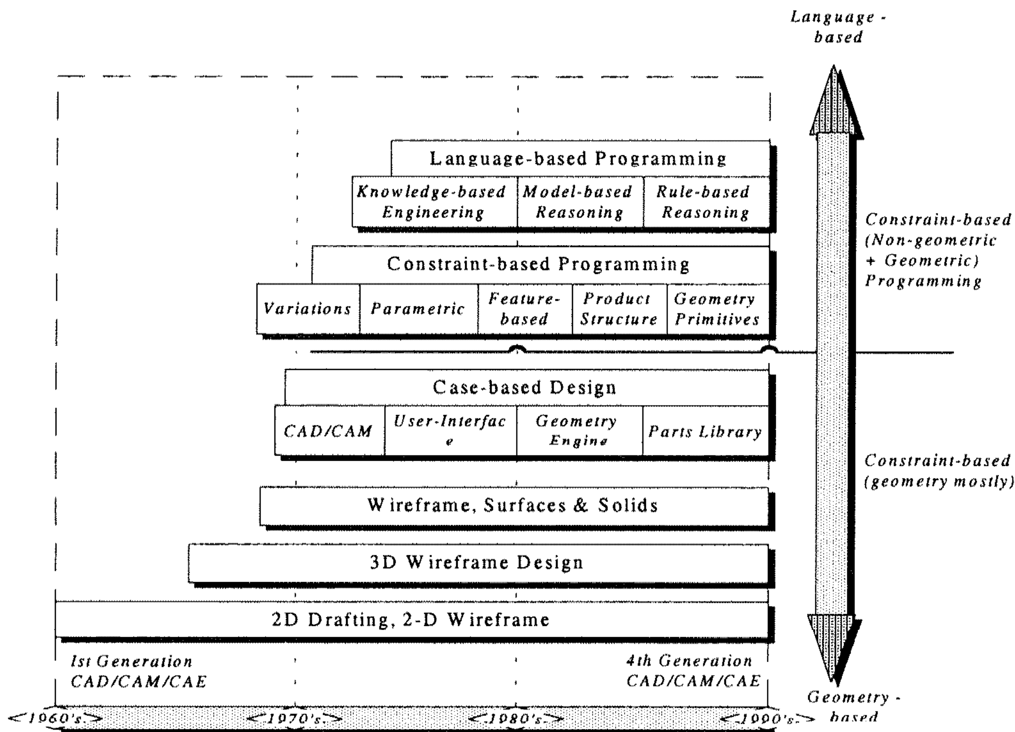


Figure 2: Evolution of Languages for Knowledge Capture

- The ***third generation*** of C4 language was constraint-based but mostly dealt with geometry. Examples include case-based design, parametric scheme, variational scheme, etc. During the period from 1980 onwards, there was a history of developments in making C4 codes more user friendly, use of a solid-based geometry engine (CSG versus B-Rep) and introduction of part library concept. There was also a flurry of activities in the use of techniques, such as parametric schemes, variational schemes, featured-based concepts for creating product structure and for defining geometric primitives [4].
- The ***fourth generation of languages***. Today is the age of fourth generation C4 languages, which is quite different from the past. Fourth generation of languages are knowledge-based techniques giving CE design work-groups the ability to capture both geometric and non-geometric information.

Languages for Life-cycle Capture = $\cup$ [Geometry-based language, Constraint-based language, Knowledge-based language]

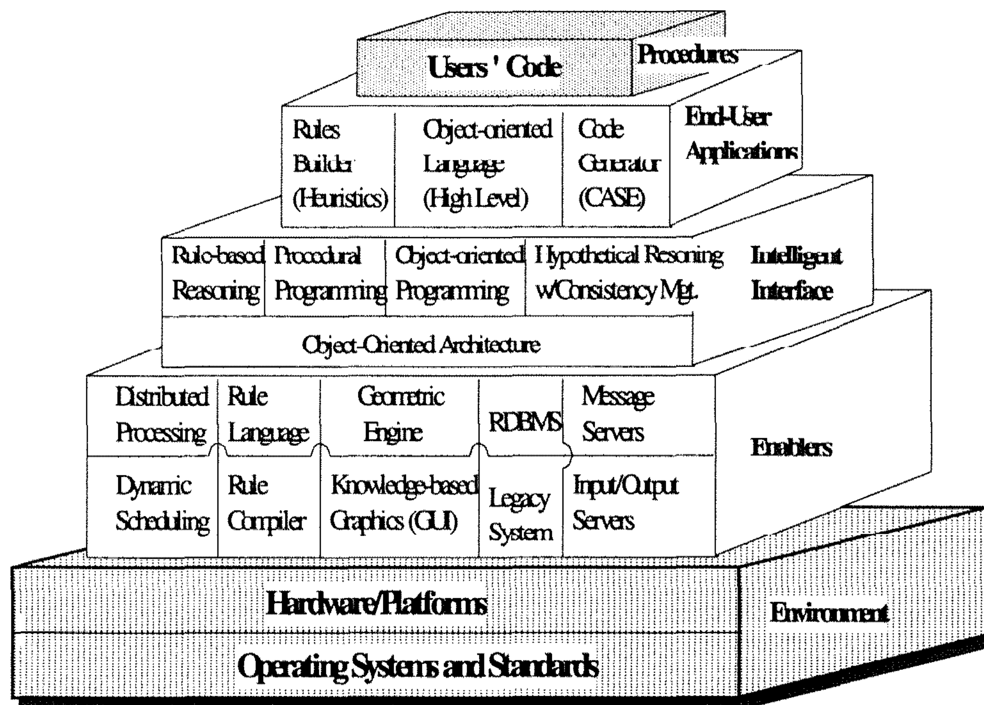


Figure 3: A Computational Architecture for a typical Knowledge-based System

3. Knowledge-based Systems (KBS)

Knowledge-based Systems (KBS) are software programs designed to capture and apply domain-specific knowledge and expertise in order to facilitate solutions of problems. Languages can be used as means to build KBS. Knowledge-based Engineering (KBE) deals with processing of knowledge. There are many ways to capture knowledge to control its processing. KBE is a process of implementing knowledge-based systems in which domain-specific knowledge about a part or a process is stored along with other attributes (geometry, form features, etc.). A computational architecture for a typical knowledge-based system is shown in Figure 3. It consists of five layers, each layer supporting the others.

- ***Environment:*** The ***first layer*** is an environment, which provides a foundation for the rest of the layers. A typical environment consists of a slew of operating systems, standards, and compute platforms (workstations, hardware, etc.).
- ***Enablers:*** The ***second layer*** consists of core enablers. Some of the key enablers included at this layer are: distributed processing, dynamic scheduling, real time rule language, rule compiler, knowledge-based graphics, GUI, geometry engine, relational database management system (RDBMS), legacy system, input/output servers, message servers, etc.

- ***Intelligent Interface***: The ***third layer*** adds intelligence to the enabling tools (second layer) and gives a programming interface to build the end-user applications. Most knowledge-based engineering tools encompass five critical technologies within an object-oriented architecture:

  (a) Rule-based reasoning
  (b) Procedural programming
  (c) Object-oriented programming
  (d) Hypothetical reasoning with consistency management
  (e) Case-based reasoning.

- ***End-user applications:*** The ***fourth layer*** is made out of end-user applications. It consists of a high level object-oriented language, a rule builder, and a code generator similar to a CASE tool.
- ***Procedures:*** The ***topmost layer*** embodies the procedures for the users' code.

3.1 Geometry-based Language

In the past, knowledge about products was mainly present in the form of geometry. Now 3-D solid geometry, as opposed to surfaces, wire-frames and other forms of geometry, is being used more often. Most traditional languages are geometry-based. They capture the attributes of solid primitives including lines and curves of a modeled object and their relationships to each other. Some high-end languages also capture information about the space inhabited by an object or about its enclosure (for example, Constructive Solid Geometry—solids). Some modelers develop complex solids by adding an extension to the traditional Boolean (join, intersect, and subtract) operations. For example, a combined solid can be driven by a 2-D sketch. As illustrated in I-DEAS Master series [8], the sketches can also be driven by geometric elements of other solids. Most solid modelers, however, fail to draw on knowledge about what the object is, its relationship to other objects or components, or its life-cycle aspects. Constraint-based CAD programs speed the design-change process by controlling and constraining object relationships based on dimensions (size, orientations, etc.), positioning, or geometrical inputs. However, such programs still focus on the geometrical aspects of the product development, not the knowledge about its life-cycle manufacture.

3.2 Constraint-based Language

Constraint-based language provides facilities for defining constraints. Most constraint-based languages provide means of incorporating arithmetic, logical functions, and mathematical expressions within a procedure. Such constraints may have a simple, linear algebraic relationship between entities to control shape (e.g., the length of line A is twice the length of line B) or geometry. The examples of geometric relationships include horizontal and vertical leveling, parallelism, perpendicularity, tangency, concentricity, coincidence, etc. Some constraints provide means to define and solve a system of linking equations that constitute a set of necessary design constraints and bounds. Finite element analysis and sensitivity

analysis are some of the options that are generally considered an integral part of a constraint-based language. Such languages encompass command structures, symbol substitution, user-written macros, control branching, matrix analysis functions, engineering data base manager and user interface to integrate complex multi-disciplinary analysis, design, and pre- and post-processing work tasks.

Finite element systems usually consist of:

(a) A set of preprocessors through which a team defines finite element meshes, applied loads, constraints, etc.,
(b) A central program that primary performs numerical computation
(c) A set of post-processors for displaying the results.

They do not provide instantiation needs. An issue often encountered in a conventionally structured program is how can CE work-groups go beyond what the FEA programs provide. If it is necessary to perform functions that are not explicit capabilities of a program, the only recourse available to the teams is a very expensive and time-consuming one. The product developer has to write a new module (in a conventional language, such as FORTRAN, C or C ++) that operates on an output data file produced by the finite element analysis program. The constraint-based language largely eliminates this difficulty, making it very easy for teams to integrate complex and highly specialized analysis and design tasks, and to create specialized input formats and output displays. Another class of constraint-based languages that use AI techniques is based on solving a constraint satisfaction problem (CSP). Formally a CSP is defined as follows [9]:

Given a set of n variables each with an associated domain and a set of constraining relations each involving a subset of the variables, find an n-tuple that is an instantiation of the n variables satisfying the relations.

In the CSP approach, most of the efforts are in the area of solving a constraint satisfaction problem automatically. Most design problems, on the other hand, are open-ended problems. They are evolutionary in nature requiring a series of frequent model updates and user interactions, such as what is encountered during a loop and track methodology (discussed in section 9, [3]). How to manage such team interactions in the CSP approach has been the topic of research in the AI community for some time.

3.3 Knowledge-based Language

In a knowledge-based engineering (KBE) system, work-group members use a design language to build a smart model of the product. Formalism for defining smart models is a knowledge-based representation paradigm for describing the life-cycle domain knowledge. KBE languages go well beyond parametric, variational, or feature-based geometry capture mode to a knowledge-based life cycle capture mode. A design work-group does not just design parts. Work-groups design products -- a collection of functional parts placed in an assembly to form a finished (that is a functional) product. A KBE language provides ways to capture geometry

and non-geometric attributes, and to write the rules that describe the process to create the assembly.

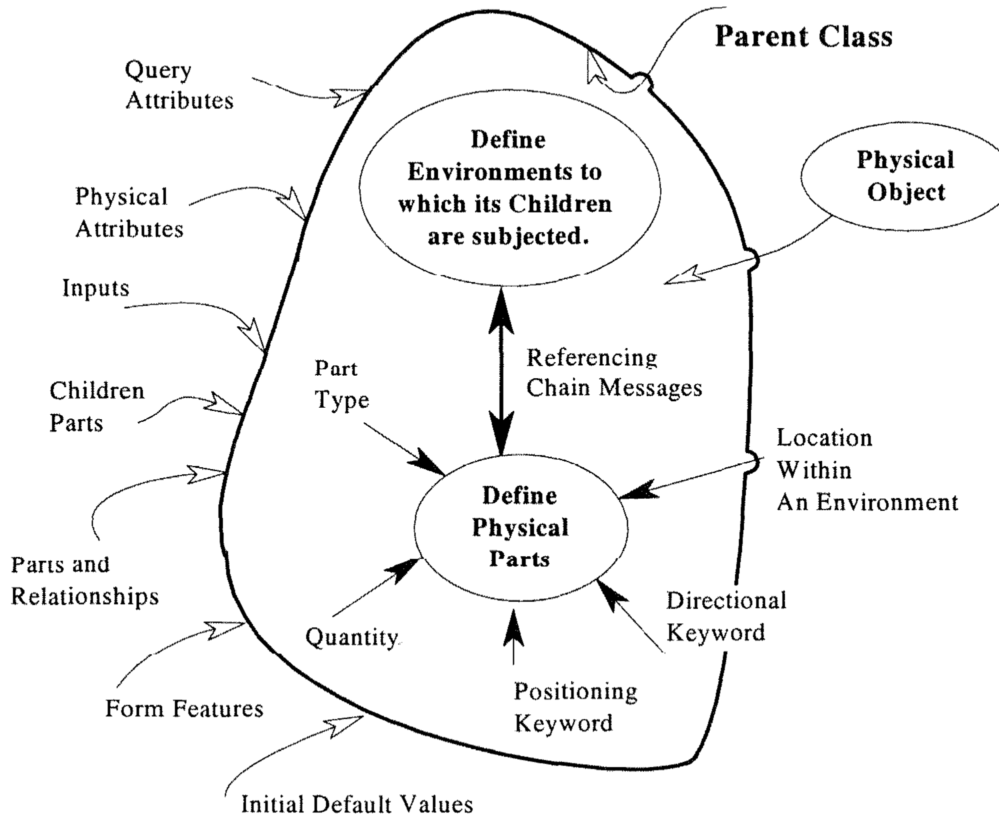


Figure 4 Definition of the "physical" Object

Such rules might include design stresses, resultant volume, or other parts' positioning, mating and orientation rules. These rules form a part of an intelligent planning procedure derived from a domain specific knowledge. The use of intelligent planning procedure in KBE replaces an exhaustive enumeration of all feasible assembly plans that would have been needed otherwise. Most of the present KBE languages use object-oriented techniques. Unlike constraint programming lan guages, which define procedures for the manipulation of objects and entities, knowledge-based languages define classes of "objects," and their characteristics and behaviors that possess built-in manipulation capabilities. KBE languages capture the totality of the functions and relationships between model elements.

The following are some of the characteristics of a knowledge-based language:

- *Object symbols or Attributes*: In a KBE language, object symbols or attributes are the backbone of the system. Attributes describe object geometry, overall physical parts, its environments, location of the parts within that environment, and any other characteristics that are required. Figure 4 shows a definition of a physical object and a few examples of some associated attributes. Some

attributes that are fixed are defined as constant-attributes. Variable attributes are design specifications whose values change. Inputs and children of an object are considered as variable attributes. Most of the physical attributes are fixed attributes. KBE languages allow a team member to define attributes in any order but they are internally recognized as "keywords." Directional and positioning keywords are keyword examples shown in Figure 4. Keywords enable "demand-driven-operations" to take place [6][7].

- *"Demand-driven" Operations*: In this mode the system determines the "order" and the "necessity" to evaluate an attribute. If a value for an attribute is demanded for the first time, the system computes the value and remembers it. In subsequent operations of the same attribute, when its value is required, the system returns the "cached" value instead of recomputing it. This method of evaluation, called "demand-driven evaluation" [6], is considered an important property for recalculating the value each time it is demanded. This type of operation relieves the programmer from assigning the order in which to evaluate the attributes. This makes programming in KBE languages significantly easier than in other languages.
- *Frame Structures with Rules*: Frames are object-oriented structures that allow for the storage of attribute information as object hierarchies. Frame representation is, thus, convenient for the storage of geometric dimensional and quantitative knowledge. Rules are used to implement the procedural expressions. The combination of an object-oriented frame structure with rules results in an adequate framework for capturing life-cycle manufacture knowledge.
- *Symbolic logic*: Symbolic logic is an underlying logic theory used in KBE to let knowledge engineers represent and manipulate the various types of knowledge required in CE. Symbolic logics are composed of object symbols (attributes), predicates, frame structures with rules, classes and instances, and kind-of inheritances. Simple logic statements can be connected using logical connectives to form compound logic statements. The set of such logic statements -- simple or compound -- is commonly called the logic theory. A full accounting for how objects and relations in the real world map to the logic symbols forms an "interpretation" of this logic theory [10].
- *Classes and Instances*: Most KBE languages allow definition of classes and instances. Classes are generic descriptions of objects, and "instances" are specific outcomes of an object-class. An object is a software packet that contains a set of related data and procedures. An object's procedures are called its methods. Objects communicate by sending messages to other objects requesting that they perform one of their methods. Object is an occurrence, or instance of a class. KBE languages often provide tools such as browser to represent objects and review instances, both graphically and non-graphically.
- *Kind-of inheritance*: The language allows definition of a "new class" from an "old-class" where the "new class" is derived from the "old class" with some "same but except" characteristics. A new class is said to inherit a portion of definitions from an existing class. Users only define the "except" changes, for example, "square" is a "kind of inheritance" from a "rectangle" object class. Inheritance allows the developer to define generalized behavior classes that can

be used by multiple, slightly different subclasses. It also allows existing classes to be extended and modified without changing the source code. This is accomplished by overriding methods at the subclass levels. It supports the creation of object models by allowing object designers or programmers to specify class hierarchies through selection of methods. The resulting object is maintained in a storage-independent form.

- *Generic Parts*: A generic part is an object-oriented structure that includes engineering rules, methods, attributes, and references to the children of sub-parts. The KBE language provides options for specifying the parts' attributes as variable attributes with no initial values specified. Other attributes are defined as a function of the variable attributes. Generic parts receive their attribute-values by means of "inputs" at run-time. The concept is useful since the generic parts' family can be replicated or instantiated at run time merely by specifying the required inputs for each part throughout an assembly. The generic parts can encapsulate other sub-parts or contain positioning or assembly information.
- *Referencing-chain*: Referencing chain is a useful concept to access an object or an attribute of a tree from any other place in the tree. It is often used to define dependencies that exist or are desired between children. An access is permitted by identifying a path that leads to the object whose attribute descriptions are required. In the definition of a physical object, shown in Figure 4, a referencing chain is shown connecting an "environment definition" with "physical parts" definitions. The concept is useful since attributes or parts can be retrieved by passing messages without actually replicating the source code, reasoning or logic behind the definition of the parts or the attributes.

Methods of Capturing Life-cycle Intent = $\cup$ [*Attribute definition, "Demand-driven" Operations, Frame Structures with Rules, Classes and Instances, Kind-of inheritance, Generic Parts, Referencing-chain*]

Depending upon the available library of primitive parts, some languages are easier than others are.

4. Creation of Smart or Intelligent Models

An important component of a smart or an intelligent model is the ability to define geometry in terms of parameters and constraints. Constraints are rules about dimensions, geometric relationships or algebraic relationships. A smart model is a reusable conceptualization of an application domain. The models contain the knowledge (attributes, rules or relations) of the application domains forming the basis for future problem solving. Rules define how design entities behave, for example, whether a hole-feature is through or blind. In the case of a blind-feature, if the part becomes thicker and the cylinder is not long enough, the hole will become a blind hole. Unlike blind-hole feature, a through-hole feature understands the rule that cylinder must pass completely through the part and will do so no matter how the part changes. There are two ways of formulating a problem that leads to smart models:

a) Constraint-based programming
b) Knowledge-based programming

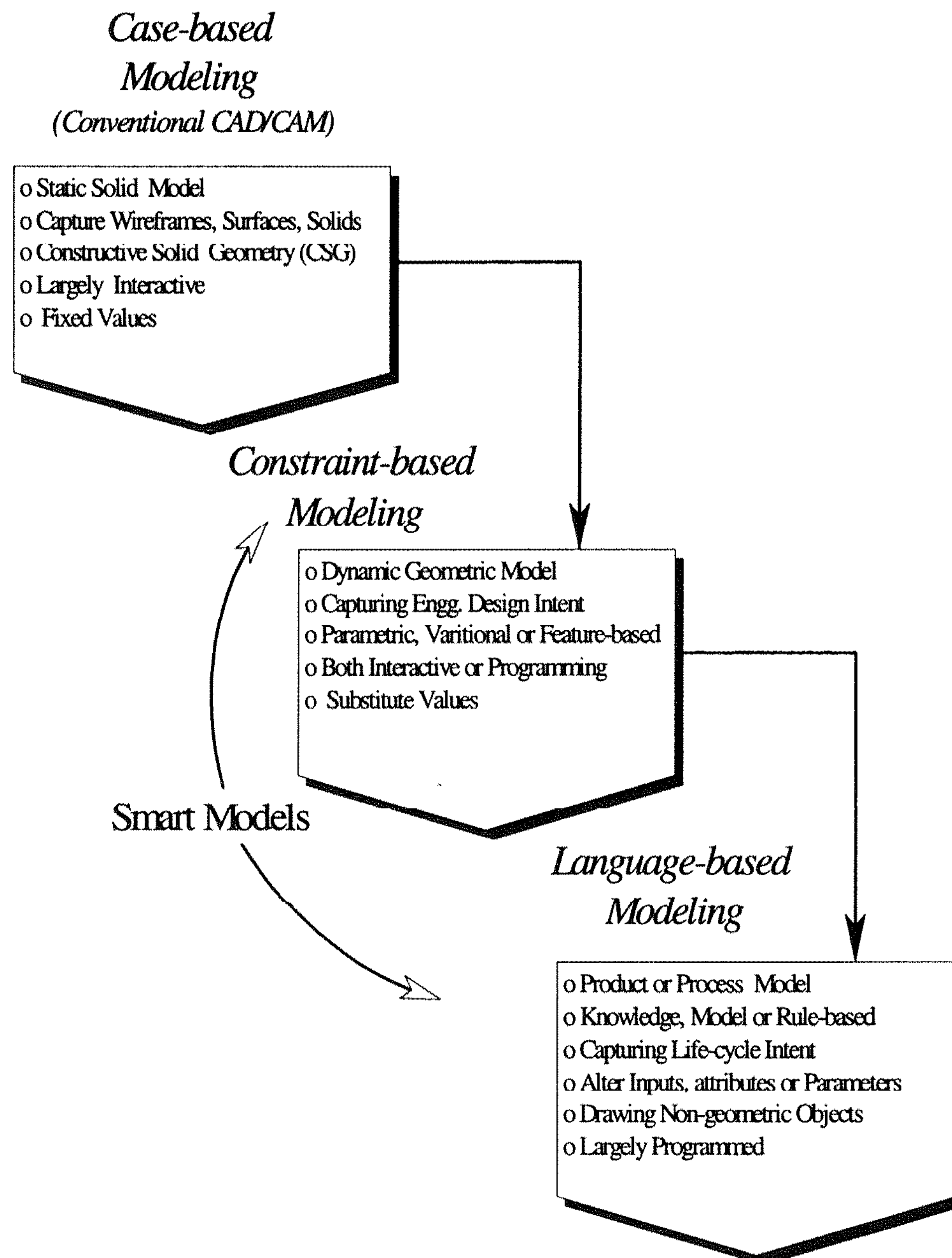


Figure 5: Properties of Conventional and Smart Models

	Representation of Parameters and Constraints	Type of Relationships between Parameters & Constraints	Method of Solving the Constraint Satisfaction Problem
Conventional Models	Non Interpretative (Fixed Dimensions)	Fixed Attributes: Points, Lines and Surfaces	Computational Geometry B-Splines, NURBS
	Interpretative (Variable Dimensions)	Variable Attributes: Points, Lines & Surfaces	Computational Geometry B-Splines, NURBS, Linear Algebra
Constraint-based Programming (Smart Models)	Parametric/Symbolic	Explicit/Algebraic	Equation Solver, Variational Geometry/Analysis
	Features/Forms	Explicit/Algorithmic	Linear/Non-Linear Programming, Optimization
	Mixed	Explicit/Algebraic + Algorithmic	Equation Solver/Linear Algebra, Multi-Criterian Optimizatiion, Optimal Remodeling
Knowledge-based Programming (Smart Models)	Rule-based Reasoning	Implicit/Heuristics	Inference Engine (Object-Oriented Programming – OOP)
	Model-based Reasoning	Implicit/Heuristics	Inference Engine (OOP)
	Case-based Reasoning	Implicit/Heuristics	Inference Engine (OOP)

Figure 6: Comparison between the Conventional and Smart Modeling Approaches.

Constraint-based modeling (CBM) or programming yields constraint-based models. Knowledge-based programming results in knowledge-based models. The major differences between the two model types (constraint-based modeling and language-based modeling) in contrast to the conventional modeling (traditional CAD/CAM system) are shown in Figure 5. In conventional modeling, the geometry is captured using a static representation of wire-frames, surfaces or solids, i.e., the geometry is captured in digitized (fixed value) form. The modeling process is largely interactive. In constraint-based modeling, since the mechanism of geometry capture is through parametric, variational or feature-based techniques, each model represents an instantiated (or dynamic) geometry. Thus, by setting new values to the CBM attributes, several instances of the geometry can be obtained. Knowledge-based modeling (KBM) is similar to constraint-based when it comes to capturing the geometry. However, because of its abilities to capture non-geometric

information and to associate rules with attributes, KBM is also suited for capturing life-cycle intent. Other contrasting features of CBM and KBM are listed in Figure 5 and further explained in the following:

4.1 Constraint-based programming

Constraint-based Programming (CBP) is a concept of formulating a problem in terms of "constraints," which may be a part of a product definition, a process definition, or an environment for the problem definition. No distinction is made between types of constraints or their sources. A spreadsheet program is a simple example of a constraint-based programming. Here equations representing constraint relationships are input to cells in a spreadsheet program. There is a close resemblance between design rationale (DR) [11] and spreadsheets. Equations that are entered into cells of a spreadsheet are analogous to "capturing" DRs, and computed cell values in the spreadsheet program are analogous to "identifications" of DRs. The cells themselves constitute the "knowledge" of CBP. If smart models are thought of as a series of spreadsheets for a concurrent team, "programming" in CBP is analogous to specifying the relationships between the cells of a spreadsheet.

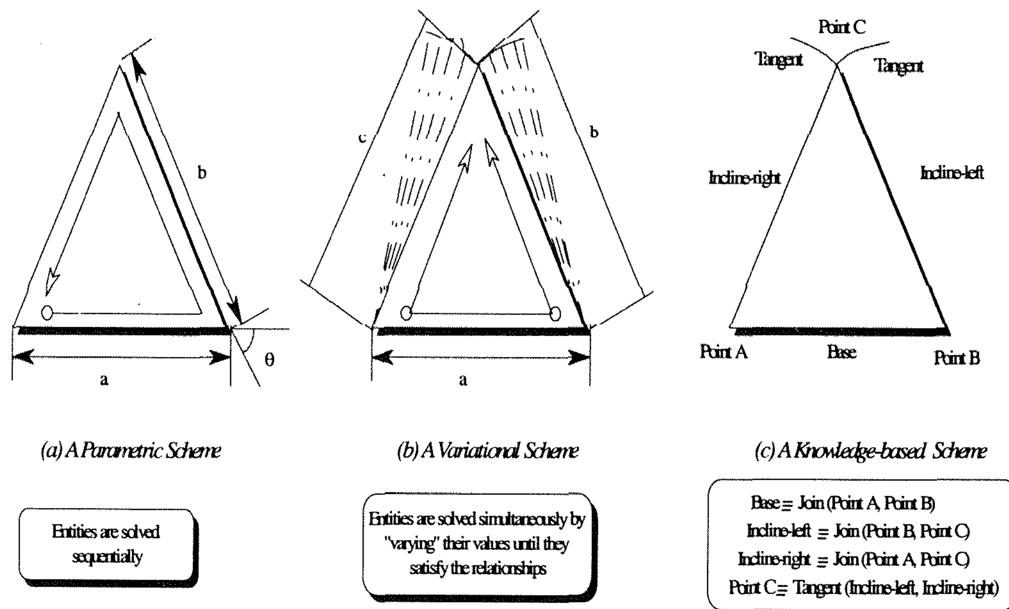


Figure 7: Difference between Parametric, Variational and Knowledge-based Schemes

The following are some typical constraint parameters that can be employed during constraint-based programming:

- ⇒ Design specifications
- ⇒ Design criteria
- ⇒ Subjective qualifications

⇒ Design constraints
⇒ Manufacturing constraints and tolerances
⇒ Material properties
⇒ Geometry
⇒ Sectional properties
⇒ Configuration and topology
⇒ Heuristics or rules
⇒ Historical data
⇒ Performance requirements
⇒ Test specifications and data

Figure 6 shows a three-way comparison between a set of key characteristics of smart models (created using constraint-based and knowledge-based programming approaches) and conventional models. Three key categories employed for comparison are: "types of representations", "types of relationships between parameters and constraints", and the method of solving the constraint satisfaction problem. They are shown in Figure 6 as columns of a matrix. The modeling categories (conventional and smart models) are listed as rows. A cell of the matrix shows the differences in the approaches used in each modeling category. In the conventional models, the geometry compatibility (such as line and arc constraints) and consistency issues are resolved through computational geometry, linear algebra, B-spline and NURBS techniques. In CBP, product design problem is defined in terms of constraints and the inter-relationships that exist between them (see Figure 7). Constraints may be a part of

- Product definition, such as geometry, materials, size, etc.
- Process constraints such as assembly constraints, tolerances, fits/clearances, etc.
- Product environment such as loads, performance, test results, etc.

In Constraint-based modeling, the constraints are of explicit/algorithmic or algebraic types. They are resolved through a set of linear and nonlinear programming, optimization and optimal remodeling techniques (see Figure 7). Such methods help develop a set of individualized criteria for life-cycle design improvement (more than what it generally appears to be the case). For example, on the surface it would appear that parametric generation of parts would not be a significant improvement over the "fixed-dimension" (static geometry) approach. In both cases, initial geometry definitions have to be captured and shared with other users of the information. The real advantages come if there is a large amount of change processing to be done to the original design. In a traditional CAD environment, this could be very time consuming and cumbersome. In knowledge-based modeling, in addition to the types of relationships specified for CBP, the constraint set also contains implicit/heuristics type of rules. Inference engines (backward and forward chaining) or constraint propagation techniques are commonly used in conjunction with object-oriented programming to resolve the

imposed constraints (see Figure 7). The knowledge-based programming approach is discussed in Section 4.2 in greater depth.

Figure 8 shows a sample set of parameters for a typical CBP environment. The types of environment depend on the descriptions of the functional intent behind the product or the process that are modeled. The key parameters surrounding a constraint-based smart model are:

⇒ Geometrical, sectional and configuration variables
⇒ Manufacturing engineering design criteria and heuristic rules
⇒ Performance requirements, cost, efficiency data, and customer satisfaction
⇒ Design specifications or constraints
⇒ Manufacturing tolerances or constraints
⇒ Material selection or qualification
⇒ Library of parts, CAD data
⇒ Historical or subjective qualifications

In constraint-based programming, algorithmic tools such as analysis, simulation, generic modeling, sensitivity, and optimization are often used as an integral part of the design process. The original set of parameters is grouped into three categories: design variables, v_i, performance functions, f_i, and design constraints, c_i. The dependencies between performance and design constraints with respect to design variables are controlled through sensitivity analysis:

$$\text{Sensitivity} = \frac{\partial f_i}{\partial v_i}$$

Additional discussion of the problem formulation can be found in Section 4.8 of Chapter 4 [12].

4.2 Knowledge-based Programming

Knowledge-based Programming is another way of creating a smart model as shown in Figure 9. In order to develop an integrated view of PD^3, which is rich and comprehensive, it is necessary to include a variety of knowledge sources and representations. Knowledge-based representations deal with *explicit* knowledge, *implicit* knowledge and *derived* knowledge.

- ***Explicit Knowledge***: Statements of explicit knowledge are available in product or process domains as retrievable information like CAD data, procedures, industrial practice, computer programs, theory, etc. They can be found both within and outside of a company. The explicit knowledge can be present as a set of engineering attributes, rules, relations or requirements. Inside knowledge deals with observations and experiences of the concurrent work-groups. Outside knowledge sources include papers, journals, books, and other product design and standard literature.

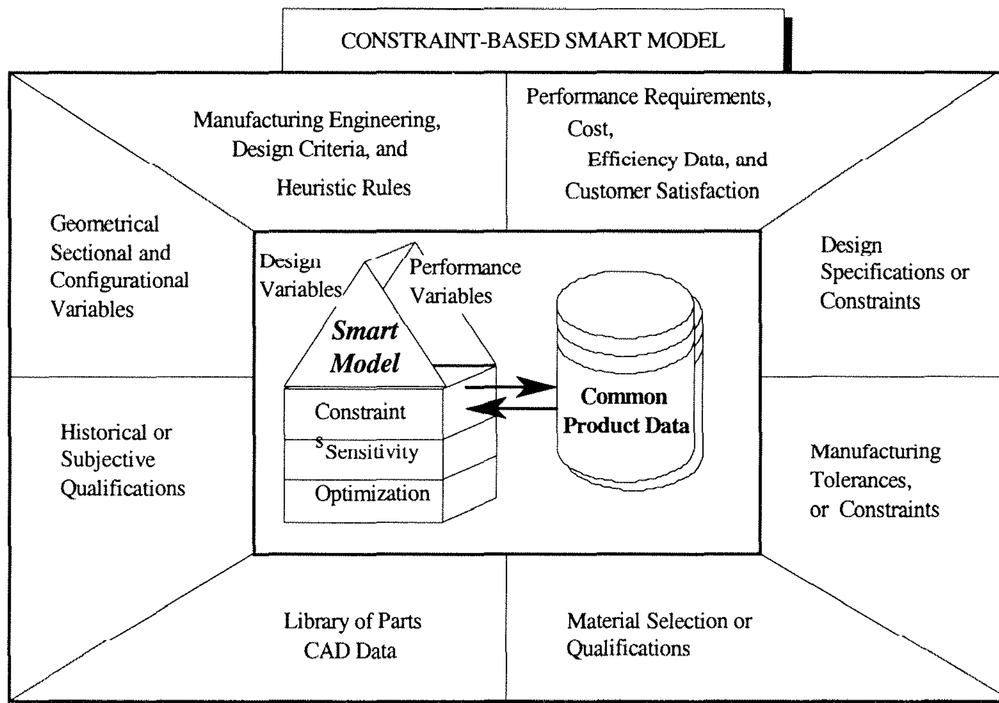


Figure 8: A Constraint-based Smart Model with Key Parameters

- ***Implicit Knowledge***: Statements of implicit knowledge are mainly available as process details such as memory of past designs, personal experience, intuition, myth, what worked, what did not, etc. Difficulties arise when such processes are obscure, e.g., intuitive or creative. Implicit knowledge includes skills and abilities of the work-groups towards the application tasks and problem solving methods. Implicit knowledge that is found outside the work-group circles is mostly in case studies and discussion dialogues. Difficulties arise when such implicit knowledge has not been articulated in a form that allows ease of use and transfer.
- ***Derived Knowledge***: Statements of derived knowledge are those that are discovered only by running external programs, such as analyses, simulation, etc. Derived knowledge is like extra- or interpolation of the current domain for which explicit knowledge is missing or incomplete. Undiscovered knowledge has been the driving force of most research and development organizations.

Knowledge-based programming software offers three basic benefits: capture of engineering knowledge, quick alterations of the product within its acceptable gyration, and facilitation of concurrent engineering.

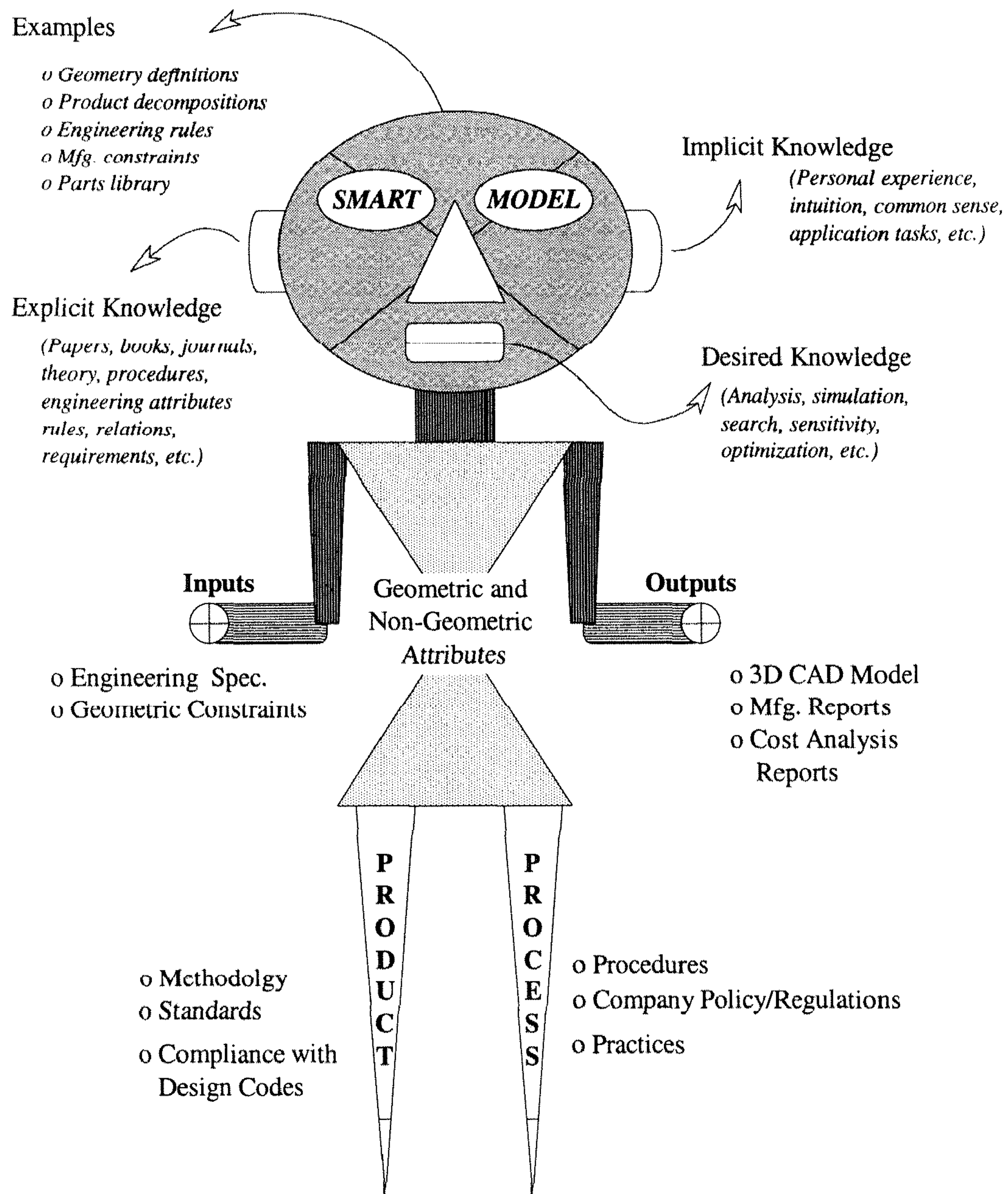


Figure 9: Salient Features of a Knowledge-based Smart Model

- The ***first*** strategic benefit is due to KBE system capturing engineering knowledge electronically. This capture allows companies to leverage scarce engineering expertise and to build on the knowledge acquired slowly over time.
- ***Second***, the system permits design variations to be generated rapidly by quickly changing a long list of inputs while maintaining model integrity.

Products designed on the KBE system can practically design their own tooling. The system also enables designs to rough out their own macro process plans automatically by drawing on knowledge for similar GT designs.

- ***Third***, KBE systems have been shown to enable concurrent engineering. Design, tooling, and process planning all benefit by working from a common integrated smart model that is able to represent, retrieve, and integrate engineering knowledge from many different sources and disciplines. Knowledge-based engineering reduces two of the most common problems that arise with team-oriented CE: boredom and time.

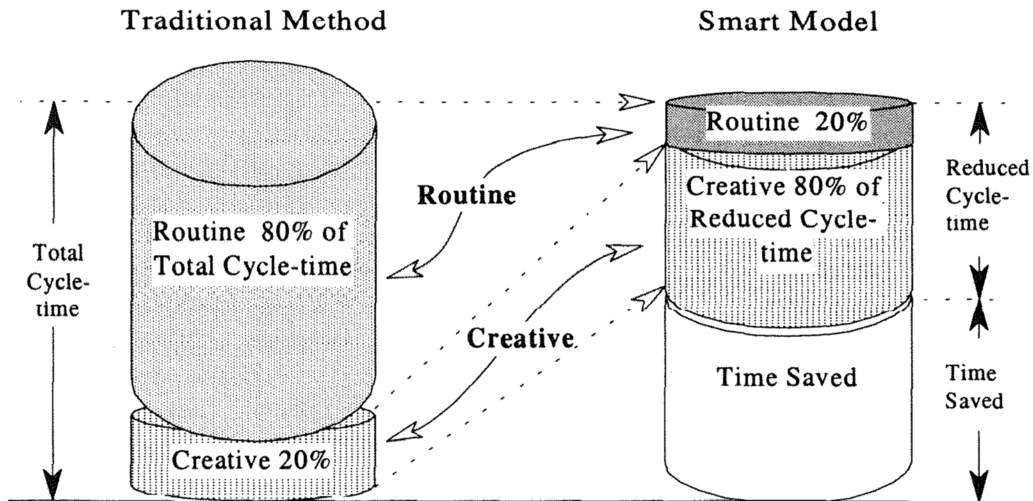
  ⇒ ***Boredom:*** Boredom crops into most traditional processes as part and parcel of their detail. Work-group members do not find it attractive to check hundreds of common drudgery details that occur every-time a new design is obtained -- from checking its specifications to tolerances—as part of a PD^3 cycle. The idea is to capture those design and manufacturing issues which impact the design of "most products, most of the time." This action is justified based on ***8020 rule***. This is commonly called the 80:20 heuristic or Pareto's law of distribution of costs [13]. In a typical situation, 80% of the assignments are routine or detail works and only 20% are creative (Figure 10). Pareto's law states that while only 20% of the possible issues are creative, they generally consume about 80% of work-group resources. The 80% of the assignments that are routine do not seem to cause any significant product's problem or consume as much resources.

  ⇒ ***Shortage of time and resources:*** The second problem is shortage of 7Ts, and resources. Many concurrent team members are not able to find enough time to devote on actual design process due to their heavy occupation with other time demanding chores such as staff-meeting, E-mail notes, management briefings, technical-walk-throughs, design reviews, etc.

KBE reduces boredom by attending to the 3Ps details in ways that reflect design procedures, standards, company policies, and compliance of design and manufacturing codes and regulations. By packaging the life-cycle behaviors into a smart model, KBE improves productivity and automation. The 80% of routine tasks is reduced to a 20% level (see Figure 10). The CE work-groups spend more time adding functional value to the design (in the saved time) rather than repeating engineering calculations for each case or recreating existing design items. If the traditional cycle-time were reduced from an initial total of 30 months to 20 months using smart models, the following calculation applies (see Table 1).

The work-group is able to concentrate more on satisfying the creative tasks up to a maximum of 80% of the reduced cycle time. They are freed from worrying about meeting the drudgery details (routine tasks) that previously took up 80% of the total cycle time in the traditional method. Thus, even after spending 40% of the work-group engineers' time more on creative tasks, there was a surplus of two months using smart models compared to the time it took following traditional method.

Time Saved = Total Cycle-time - Reduced Cycle-time



Time Saved = Total Cycle-time - Reduced Cycle-time

Figure 10: Key Benefits of Knowledge-based Engineering

Table 1: Comparison of Savings

Actions or tasks	Traditional Methods	Smart Models	Remarks
Total time taken to finish the tasks	30 months	20 months	Assumptions
Ratio of time spent doing routine/creative tasks in months (percentage)	24/6 months (80%/20%)	4/16 months (20%/80%)	Following the definitions of smart and traditional models
If 40% more time is spent doing creative tasks then number of extra months needed	12 months (40%)	8 months (40%)	This shows how you can do more with less.
Difference in time compared to traditional Method	12 Months (deficit)	2 months (surplus)	

5. Discussion and Future Trends

Depending upon the frequency and need for creating modified designs, it is desirable to weigh the benefits. Alternatives are either to apply an additional effort and the time required to develop the smart models or to carry on the design in a traditional way. Knowledge-based development of design may not be worth the effort if each design is a unique design and significant changes to the product over its life cycle are not expected. However, this perspective changes quickly if one attempts to view the worth with respect to the overall company performance. Techniques like parametric, feature-based and knowledge-based models all facilitate Concurrent Engineering and Collaborative Engineering. Some techniques offer better capabilities than others do. For example, parametric or feature-based techniques can change the geometry of the design very quickly. However, in doing so, design work-groups no longer have the assurance of knowing if all, or indeed, any of the non-geometric (e.g., engineering and manufacturing rules) have been violated. Through Knowledge-based engineering, one can capture, besides parametric geometry, the engineering and manufacturing rules for geometric modifications. When the specifications demand a new geometric design, the corresponding rules are automatically engaged to meet the engineering and manufacturing requirements and to achieve the best possible compromise. An interesting aspect of this approach is that one can capture and build trial processes (such as levels of analysis iterations, sensitivity, optimization, etc.) into a TPM in order to establish the best design. The effect of these trials is to automatically run thousands of analysis iterations in the background before the final design is selected. All of this can be transparent to the work-group members, who simply want to feed in specifications and are interested in reviewing the outcome that works. Most of the smart techniques provide some form of an electronic control over the design process which can afford a dramatic reduction in change processing such as design revisions, design changes, etc. The term automatic generation is construed as computer-aided generation of outputs such as design renderings, process plans, bill-of-materials, numerical control instructions, software prototyping, machining, replacement parts, product illustrations, etc. If an electronic capture of design intent is extended to include automatic or partially automatic design generation, the worth of electronic capture of design increases substantially. Furthermore, if such generation is done in 3-D solids that do not require assembly information in downstream processing, electronic capture becomes almost an irrefutable requirement of the product definition process.

In recent years use of smart models are increasing. More and more product development teams are capturing product and process-related knowledge. While their usage is increasing there is this need to make the captured knowledge in smart models used (leveraged) more widely across an enterprise. To facilitate such "knowledge reuse" product development teams are following standard techniques and tools for building such smart models. Thus, creation process for building Smart models is becoming more and more structured. The four main elements that make a smart model more structured are:

1. ***Objects, independent problem domains and tasks:*** Objects and the system are partitioned into a number of relatively independent loosely coupled problem domains so that the decomposed tasks are of manageable size. Each sub-problem can be solved somewhat independently from other domains and each can be performed in parallel. This is discussed in Chapter 4 -- "Product Development Methodology [12]."
2. ***Design Rationale:*** Irrespective of the modeler used for creating smart models (knowledge-based modeler or constraint-based modeler), the modeler splits the data into *rule-sheets* and *frame-sheets*. The rule-sheets contain the equations, and the variables are automatically placed in the frame-sheets. Additional sheets can be defined as needed, such as data settings, constraints-sheet and tables where results are tabulated, and plot-sheets which define how results are plotted. This gives the design problem a symbolic structure rather than a numerical processing structure. The other advantages of using design rationales are [11]:

   ⇒ Definition of objectives and constraints are in terms of problem parameters, design rules, or intent with possibly incomplete data.
   ⇒ DRs establish a set of relationships (explicit or implicit) between parameters and constraints.
   ⇒ DRs capture the informal heuristics, or chains of reasoning, rather than a set of well defined algorithms.

3. ***Method of solving the constraint based problem***: This is discussed at greater length in Chapters 4.1 and 4.2 of Concurrent Engineering Fundamental Book [12].
4. ***Databases, Technical memory or knowledge-bases***: Databases, technical memory or knowledge-bases derived from a surrogate product can serve as a basis for developing smart models and conducting strategic studies [13]. When actual product data is not available, a surrogate object can take the place of a technical memory. Later, when actual data becomes available, the information in technical memory is replaced on a modular basis and the information model can be updated dynamically.

6. Concluding Remarks

Knowledge-based programming (KBP) provides an environment for storing explicit and implicit knowledge, as well as for capturing derived knowledge. When these sets of knowledge are combined into a total product model (TPM), it can generate designs, tooling, or process plans automatically. Unlike traditional computer-assisted drafting (e.g., a typical CAD) programs that capture geometric information only, knowledge-based programming captures the complete intent behind the design of a product — "HOWs and WHYs," in addition to the "WHATs" of the design. Besides design intent there are other knowledge (such as materials, design for X-ability, 3Ps, process rules) that must be captured. Knowledge-based Engineering (KBE) is an implementation paradigm in which complete knowledge about an object (such as a part) is stored along with its geometry. Later when the

part is instantiated, the captured knowledge is utilized to verify the manufacturability, processiability and other X-abilities concerns of the part. One important aspect of knowledge-based engineering is the ability to generate quickly many sets of consistent designs instead of just capturing a single idea in a digitized CAD form that cannot be easily changed. Knowledge-based programming technology encourages development of a "generic" smart model that synthesizes -- what is needed in many life-cycle instances in complete detail. This is a most flexible way of creating many instances of a model, each being a consistent interpretation of the captured design intent. The interpretation is the result of acting on rules captured through the smart models by feeding in the specific inputs at the time of the request.

The current trend in smart models creation is to:

(a) Use some structured process for capturing the knowledge content and
(b) Store those rules and knowledge outside the smart models in some neutral object-oriented databases.

This way others could access them in more places, if they need them, during the product development across an enterprise.

References

[1] Nevins, J.L. and D.E. Whitney, eds.,1989, *Concurrent Design of Products and Processes*, New York: McGraw-Hill Publishing, Inc.

[2] Kulkarni, H.T., B. Prasad, and J.F. Emerson, 1981, "Generic Modeling Procedure for Complex Component Design", *SAE Paper 811320*, Proceedings of the Fourth International Conference on Vehicle Structural Mechanics, Detroit, Warrendale, PA: SAE.

[3] Prasad, B., 1996, Concurrent Engineering Fundamentals: Integrated Product and Process Organization, Volume 1, Upper Saddle River, NJ: Prentice Hall PTR, Inc.

[4] Finger, S., M.S. Fox, D. Navinchandra, F.B. Prinz, and J.R. Rinderle, 1988, "Design Fusion: A product life-cycle View for Engineering Designs," Second *IFIP WG 5.2 Workshop on Intelligent CAD*, University of Cambridge, Cambridge, UK, (September 19-22, 1988).

[5] Whetstone, W.D., 1980, "EISI-EAL:Engineering Analysis Language", Proceedings of the Second Conference on Computing in Civil Engineering, New York: American Society of Civil Engineering (ASCE), pp. 276-285.

[6] Rosenfeld, L.W., 1989, "Using Knowledge-based Engineering," *Production*, Nov., pp. 74-76.

[7] ICAD, 1995, *understanding the ICAD Systems*, Internal Report, Concentra Corporation, Burlington, MA.

[8] IDEAS Master Series, 1994, Structural Dynamics Research Corporation, SDRC IDEAS Master Series Solid Modeler, Internal Report, Version 6, 1994.

[9] Kumar V., 1992, "Algorithms for Constraint-Satisfaction Problems: A Survey", *AI Magazine*, Volume 13, No. 1.

[10] Winston, pH., 1992, *Artificial Intelligence (3rd Edition),* New York: Addison-Wesley Publishing Company.
[11] Klein, M., 1993, "Capturing Design Rationale in Concurrent Engineering Teams," *IEEE Computer*, Jan. 1993, pp. 39-47.
[12] Prasad, B., 1997, Concurrent Engineering Fundamentals: Integrated Product Development, Volume 2, Upper Saddle River, NJ: Prentice Hall PTR, Inc.
[13] Nielsen, E.H., J.R. Dixon, and G.E. Zinsmeister, 1991, "Capturing and Using Designer Intent in a Design-With-Features System," Proceedings, 1991 ASME Design Technical Conferences, 2nd International Conference on Design Theory and Methodology, Miami, FL.