1992

A graph-based knowledge representation to support design abstraction in CAD systems

Simon Szykman
Carnegie Mellon University

Jonathan Cagan

Carnegie Mellon University.Engineering Design Research Center.

A Graph-Based Knowledge Representation to Support Design Abstraction in CAD Systems

Simon Szykman, Jonathan Cagan

EDRC 24-81-92

A Graph-Based Knowledge Representation to **Support** Design Abstraction in CAD Systems

Simon Szykman
and
Jonathan Cagan

Department of Mechanical Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Electronic Mail: szykman@edrc.cmu.edu
jcagan@edrc.cmu.edu

Abstract. This paper introduces a computational framework to support the process of design abstraction. A domain-independent graph-based representation is presented to support decomposition of a design problem while modeling coupling between subproblems, multiple representations of a design at varying levels of abstraction, and analysis of a design. The representation separates knowledge into knowledge about the design instance, the domain of design, and the various levels of abstraction. The use of the representation is illustrated with two examples: the design of a two-tiered column for buckling and the design of a metal rolling operation for a continuous casting manufacturing process.

1 INTRODUCTION

This paper introduces a domain-independent graph-based representation to support abstraction of design knowledge for conceptual design By *abstraction* we mean the removal of a portion of detail to reduce the amount of information needed to reason about a complex problem. Representation of knowledge[1] for abstraction through the use of computer aided design (CAD) systems is thus an important area of current research.

The overall goal of this research is to provide a framework upon which a CAD system for conceptual design can be built Abstraction is often used by designers when faced with this complex design task because their short term memory *(i.e.,* working memory) has a limited capacity (Miller, 1956). Abstraction allows designers to increase the amount of information in their working memory (Newell and Simon, 1972), thus enabling them to have a broader perspective of the current state of a design. Abstraction is also linked with the use of analogy, which takes place at high levels of abstraction and plays an important role at the conceptual stage of the design process (Gordon, 1961). For a general overview of abstraction in engineering design and a review of related literature, see Hoover, era/., (1991).

Paz-Soldan and Rinderle (1989) observe that conceptual design involves alternating steps of reasoning in which detail is initially ignored to focus on one design aspect, followed by the addition of complexity to the design representation. This addition of complexity enables the engineer to evaluate the design and/or determine die sensitivity of the design to parameter variations. They call this iterative approach a strategy of *alternate abstraction and refinement.* It is therefore important that a knowledge representation used for a CAD tool be able to simultaneously maintain representations of a design (or portions of a design) at different levels of abstraction. An approach to creating these levels is through problem decomposition.

Decomposition of a design allows the engineer to reduce a problem into a series of subproblems which can, to some extent, be solved independently. However, even when a design problem can be decomposed, the resulting subproblems can not always be decoupled. In other words, although the subproblems can be considered independent for convenience, die effects they have on one another may be significant. This is particularly true for mechanical design, due to the complex behaviors and interactions which are characteristic of mechanical devices (Rinderle, 1986). Therefore, a CAD tool should support decomposition, but the knowledge representation must be able to model these interactions.

By progressing from the abstract levels at which conceptualization takes place to more detailed levels, a design concept can be evaluated through analysis in one or more engineering disciplines. We will refer to engineering disciplines, such as beam mechanics, vibrations, etc., as *domains.* To allow analysis in more than one domain, a design should be represented in a domain-independent manner. Clearly, information about a design is required for analysis in a specific domain. However, the since equations for performing an analysis are general domain equations, they should not explicitly be pan of the representation of the design itself. By keeping the design representation separate from domain knowledge, multiple analyses can be performed on an particular design concept by accessing the appropriate domain knowledge when needed.

---

1 The words *knowledge* and *information* are used interchangeably in this paper.

Human-computer interactions are a necessary part of CAD. This is even true for intelligent CAD tools, which are systems capable of carrying out part of the design process using some form of qualitative or quantitative reasoning (Tomiyama, 1990). Motivated by this, our approach towards knowledge representation is to maintain consistency with the way designers organize their ideas. Some key points related to this approach are:

1. *Human designers use abstraction.* There is a need for management of the different levels of abstraction, as well as a method of keeping track of the relationships between them.
2. *Designers decompose problems.* Supporting decomposition requires representing not only the components of a design, but also the interactions between them. Allowing coupling between components is an important aspect of design representation.
3. *Designers generate and test designs.* Once a design concept has been represented, it may need to be evaluated This requires that the representation support analysis of a design.
4. *Using the representation should be as simple as possible.* The procedure for creating, modifying and analyzing designs should be simple, allowing the engineer to only be concerned with the design problem.

In the next section, we introduce a graph-based knowledge representation which is designed based on the above requirements. In section 3, the framework is used to design part of a continuous casting manufacturing process. Section 4 then discusses an object-oriented implementation.

2 THE GRAPH-BASED REPRESENTATION

In this section, the graph-based representation and its associated structure are formally presented and defined. We propose a framework which allows relevant design knowledge to be represented at multiple levels of abstraction and to be modified as the design changes. First a knowledge classification is introduced which permits the separation of domain and design instance information from design topology. Then the formal graph representation is introduced.

2.1 Knowledge Classification

To represent a design, one needs to be able to accurately describe the various parts of the design, the connections between these parts, and boundary conditions applied to the design. To do this adequately, one must define the topology of a design *(i.e.,* how different parts of the design are connected), as well as parametric or qualitative information about the design, such as mass, material properties, stiffness and damping information, and boundary condition types.

Topolog: a often be thought of independently from parametric information. This is convenient for several r. ns. At the early stages of design, the topology may be fixed before the parametric information i .nown. For example, in designing a tow truck winch, an engineer may determine the configuration *of* the design (a cable attached to a shaft, turned by a motor) before assigning any parameters (such as type of motor, shaft diameter, cable thickness, or dimensions). In addition, because abstraction is the result of ignoring a certain amount of detail to reduce problem complexity, parametric information is often not needed for the abstract representation of a design.

The most important representation issue related to analysis is the independence between domain knowledge and knowledge about the design instance. We require that the representation of the design be separate from **the** domain knowledge. This achieves the desired generality, by enabling a single design representation to be used for analysis in multiple domains. A second advantage is that because domain knowledge is typically applicable to a class of problems, it can be used to analyze several designs once it has been represented

Finally, data management issues arise from supporting decomposition and abstraction. The knowledge representation requires a way of keeping track of die multiple representations of a design at different levels of abstraction, and relating design representations resulting from problem decomposition to one another.

To satisfy these needs, we distinguish between three categories of knowledge in this representation (see Figure 1). First, there is information about a design instance. This is referred to as *system-specific* knowledge. System-specific knowledge is used to create a representation of the system of interest, and contains *no* general domain information. System-specific knowledge is further divided into two categories: *connectivity* knowledge and *non-connectivity* knowledge. Connectivity knowledge is used only to describe the topology of the system, that is, which regions are connected to which others, in what directions, and where boundary conditions are applied. All other knowledge about the system, such as the types of parametric information described above, falls under non-connectivity knowledge. To illustrate the distinction, consider the design of a truss. Connectivity knowledge describes where beams are connected, in which directions they lie, and so on. All information about the truss which is not directly related to the topology, such as material properties, beam geometry, and moments of inertia, is considered to be non-connectivity knowledge.

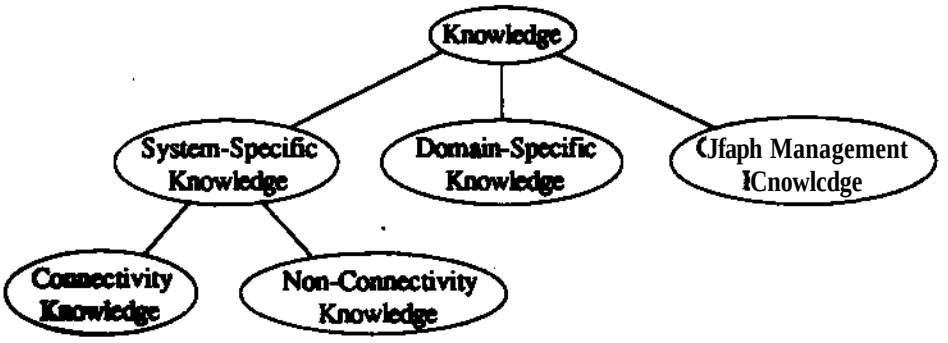

Figure 1. Types of Knowledge

The second type of knowledge is *domain-specific* knowledge. This knowledge consists of only general domain information (such as equations based on first principles or design heuristics) and contains no information about the *instance* of the design which is being represented. Domain-specific knowledge is typically applicable to a class of problems, not just the one under consideration. The equations which form domain-specific knowledge are expressed in terms of variables. Particular values of those variables are properties of an instantiated design, and are therefore considered system-specific rather than domain-specific knowledge. Returning to the truss design, the domain-specific

knowledge consists of the general information used in the analysis of trusses, such as force-displacement relations or stress equations.

The thin! type of knowledge is *graph management* knowledge. Graph management knowledge is used to organize **the** various levels of the design, such as the different components in a decomposition for different levels of abstraction. This is done with the use of *subgraphs.* A subgraph is a graph which is related to another graph (or part of another graph) in a user-specified way. These knowledge types will be described in more detail in subsections 2.3 to 2.6.

2.2 **The Graph** Structure

Given this classification of knowledge and separation of domain information from topology, we now introduce a graph-based representation that supports design abstraction by maintaining the topological relations and associating them with pertinent domain and design instance knowledge. Knowledge is stored in a generic form using data structures called *knowledge modules,* which associate knowledge with a particular part of a design. Cagan (1991) discusses the concepts of knowledge modules in greater detail.

We define an *object-module tuple* as an *object* with pointers to a *system-specific knowledge module* (SSKM) and a *domain knowledge module* (DKM). An object is a data structure consisting of a list of pointers to other objects and to knowledge modules. System-specific and domain knowledge modules are data structures used to store system-specific and domain knowledge. The four types of objects are ***node* objects, *boundary condition* objects, *link* objects, and *graph management* objects. For** convenience, the node, boundary condition and link objects, will be referred to as nodes, boundary conditions and links. Each object can point to a system-specific knowledge module and a domain knowledge module. If an object has no system-specific or domain knowledge module, the object points to *nil,* instead of to a knowledge module.

Using the convention that bold letters represent sets, a graph G can be formally defined as:

$G = \{SG, N, BC, L, GM\}$,

where

SG « {SGi,..., SGi} is a set of subgraphs *(i.e.* a graph can be composed of other graphs),

N = {Ni,..., Nj) is a set of *node object-module tuples,*

BC = {BCi, .-, BCk) is a set of *boundary condition object-module tuples,*

L = {Li,..., Lm) is a set of *link object-module tuples,*

GM = {GMi,..., GM_n) is a set of *graph management object-module tuples,*

One or more of the sets SG, N, BC, L and GM may be empty, meaning that a graph can, but does not have to include every type of component

The various components which form a graph will now be described in greater detail. A simple example will be utilized to illustrate their use. A more complex example will follow in section 3 Figure 2 shows a pair of slender columns (the top one pinned and the bottom one clamped) in compression. If the load on the columns exceeds a critical load, the columns will fail by buckling

Note that the two columns can be individually modeled (decomposed), however their behavior is coupled in that a change in one column may affect a change in die other column.

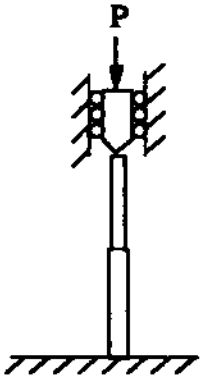

Figure 2. Buckling Columns

Figure 3 shows the graph which represents the system. Note that the graph consists of seven object-module tuples, and that the topology of the system (the connectivity knowledge) is defined by how the object-module tuples are connected. The non-connectivity knowledge and the domain knowledge are stored in the system-specific and domain knowledge modules, respectively. Objects which have pointers to *nil* are objects which do not have SSKMs and/or DKMs. Every object **and** knowledge module has a label, or name, used for identificaticm purposes. These labels are arbitrary and are assigned by the designer. The various components which form the graph in Figure 3 will be discussed in the following subsections.

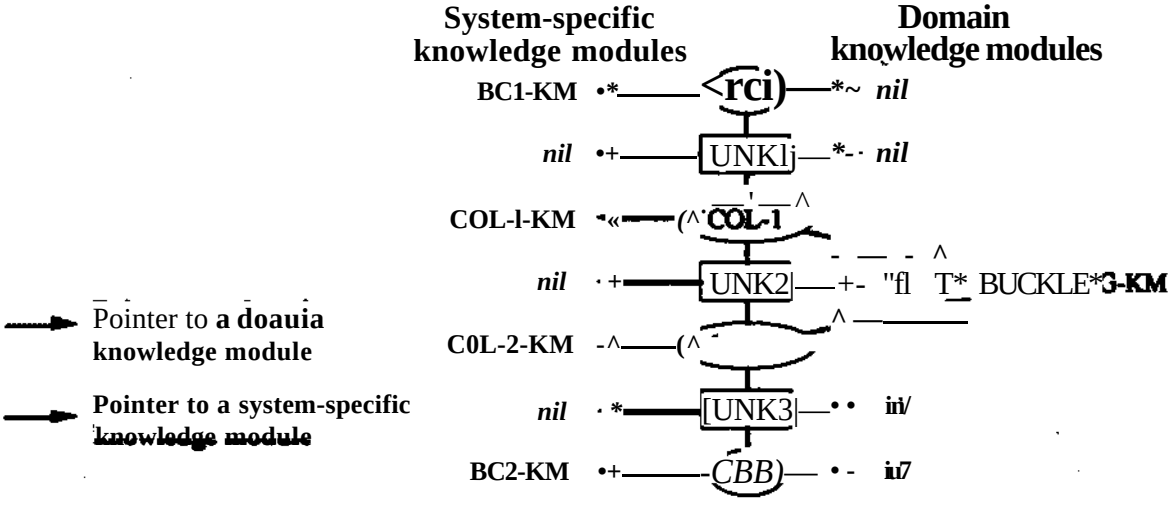

Figure 3. Graph for the Buckling Example

2.3 System-Specific Knowledge

System-specific knowledge consists of information about a particular instance of a design. System-specific knowledge is represented using the sets of object-module tuples N, BC and L. Nodes are used to represent *regions (i.e.,* physical parts of a design), boundary conditions represent a boundary

condition **applied to the system,** and links represent connections between regions and/or boundary conditions. **As ilhtttrated above,** connectivity knowledge is defined by the manner in which object-module tuples **are** connected Non-connectivity knowledge associated with each node, boundary condition **or link** is **stored** in its system-specific knowledge module. These SSKMs take the form of a list of variables with values. Since non-connectivity **knowledge** can be qualitative, these values are not necessarily numerical.

The following subsections describe nodes, boundary conditions, links and their associated SSKMs in greater detail, using the buckling columns as an example. In addition to representing system-specific knowledge, the context in which the system is being examined must be defined by adding domain knowledge. This is done by pointing to a DKM. Domain-specific knowledge will be described in subsection 2.4.

2.3.1 Node Objects and Associated System-Specific Knowledge Modules

A node is a data structure which represents a physical part of the system. A node object-module **tuple** is a node which points to a SSKM and a DKM. A node can also point to links (which are used to connect the node to other nodes or boundary conditions), and a graph management object A **SSKM** associated with a node contains all non-connectivity information of interest for the region represented by that node.

The first step in modeling our example system is to represent the two columns. For the purpose of a buckling analysis, each column is considered to be a single region and can therefore be represented by one node. Two nodes, arbitrarily called COL-I and COL-2 are created. Two SSKMs called COL-I-KM and COL-2-KM are created, and "connected" to their respective nodes using a pointer (see Figure 3). The information contained in each SSKM is the value of E (the elastic modulus for the column material), I (the cross-sectional moment of inertia), and L (the length), associated with that column.

2.3.2 Boundary Condition Objects and Associated System-Specific Knowledge Modules

A boundary condition is a **data** structure which represents a boundary condition in the system. A boundary condition **object-module** tuple is a boundary condition which points to a SSKM and a DKM. A boundary condition must point to a link, which connects the boundary condition to the node at which that boundary condition is applied

For our buckling example, there are two boundary conditions, one for each column. The boundary conditions, called BCI and BC2, each point to SSKMs, respectively called BCI-KM and BC2-KM (sec Figure 3). The information contained in each SSKM is the type of boundary condition at the end of the column. In this case, referring to Figure 2, the boundary conditions are a pinned end at the top and a clamped end at the bottom.

2,3.3 Link Objects and Associated System-Specific Knowledge Modules

A link is a data strtKtwe which is used to define a connection between two objects - either two nodes, a node and a boundary condition, or two boundary conditions. A link object-module tuple is a link which points to a SSKM and a DKM. Note that information contained in the SSKMs associated with links is limited to characteristics of the *connections* between the objects which are linked together, and not information about die nodes and/or boundary conditions themselves.

Three links are necessary for our example system (see Figure 3). links LKNKI, UNK2, and UNK3 are created UNK2is used to connect the two nodes to one another, and the other links are used to connect each node to a boundary condition. For our buckling example there are no SSKMs associated with the links because no information about the connections is relevant to the analysis; the buckling analysis implicitly assumes a rigid connection which transfers coupling information between the individual columns. Note that the links in Figure 3 point to *nil* instead of to SSKMs.

Links are not confined to only representing physical connections. In real systems, there is often a cause-and-effect relation between parts of the system which are not physically connected. One example of these connections (referred to as *causal* connections) is dynamic coupling where the motion of a body can affect the motion of another body even though the bodies are not directly connected to ene another. Another example of causal connections is distance forces, such as gravity or magnetism, which occur without a physical connection. Links can be used to represent these causal connections*

2.4 Domain-Specific Knowledge

Domain-specific knowledge, as stated previously, consists of general information relevant to a particular domain. Domain-specific knowledge can be used to model a *class* of designs via generic design equations, which are contained in domain knowledge modules. To be applicable to a variety of designs within a given domain, the domain knowledge module must not contain any system-specific knowledge. Because the equations may apply to a more than one object, several objects may point to the same DKM. Thus, although there is a one-to-one correspondence between objects and SSKMs, the correspondence between objects and DKMs may be many-to-one.

DKMs differ in structure from the system-specific knowledge modules which consist of lists of variables and their values. The domain equations are in terms of variables but the *values* of those variables for a *particular* system are system-specific knowledge, and are stored in SSKMs. The domain equations are therefore incorporated as part of a piece of code. The purpose of this code is to search the graph to identify which objects contain what information in their knowledge modules, retrieve the values necessary for a particular analysis, and substitute those values fen* the variables in the domain equations.

For the analysis in our example, the domain knowledge module is called BUCKLING-KM. This knowledge module contains a function which evaluates the theoretical value of the end-condition constant (based on the boundary conditions), and Euler's formula for the critical buckling load:

$$P_{cr} = n\pi^2 EI/L^2 \qquad (1)$$

where E is the elastic modulus of the column material, I is the moment of inertia of the column, L is die column length **and n** is **the** end-condition constant This DKM can be used for a class of buckling problems, where **tfcf** number of columns, the column material, length or cross-sectional geometry may all vary. **The loadtlfi an** individual column can be calculated by adding the weights of the columns above to **the external** load; these influences from the other columns can be derived through the link connections. Note that in Figure 3, only the nodes point to the DKM, because they represent the columns being analyzed

2.5 The Buckling Example Graph

Returning to the set fonnalization, we can represent the graph for the buckling example as

G = {SG,N,BC,L,GM},

where

SG = { } is the set of subgraphs,

N = (C»l^l^COL-l-KM,BUCKLING.KM), COL-2-KCX)L-2-KM,BUCKLING-KM)} is the Set of node object-module tuples,

BC = {BCI-KBCI-KM, *nil),* BC2-^BC2-KM, *nil)}* is the set of boundary condition object-module tuples,

L = {UNKi-Km', niQ, UNK2-Km', *nil),* UNK3-*(m', *nil))* is the set of link object-module tuples,

GM = { } is the set of graph management object-module tuples.

Empty brackets indicate an empty set and an arrow indicates an object pointing to a pair of knowledge modules (a SSKM and a DKM), forming an object-module tuple. Note that as shown by the pointers to *nil,* the links in L have no SSKMs, and neither the links in L nor the boundary conditions in BC point to a DKM. This example has no subgraphs or graph management object-module tuples. These concepts will be defined in the following section.

2.6 Graph Management Knowledge

A system can often be decomposed into numerous subsystems. These subsystems may be addressed independently, but are usually coupled to one another at some level. Even if the subsystems are coupled, their decomposition may be convenient for the designer, particularly when dealing with complex systems. **These** subsystems are represented using *subgraphs.* A subgraph is a graph which is related to another **graph,** or part of another graph. Graph management object-module tuples are used to implement the concept of subgraphs.

A graph management object-module tuple is a graph management object which points to a graph management knowledge module (so-called to distinguish it from SSKMs which contain system-specific knowledge). Because graph management objects have no domain knowledge, they always point to *nil* for a DKM. A graph management object must point to two nodes: a *parent node* and a *subgraph node.* The parent node is the node which has a subgraph and the subgraph node is a node in the subgraph. Graph management knowledge modules also have a structure which is different from

that of die system-specific knowledge modules. A *graph* management knowledge module is used to identify tj [illegible]pe e n a node and its subgraph. The use of graph management object-module tqples to * Pt e graphs to one another is illustrated in Figure 4, where Graph B is a subgraph of Node 1 in Graph / #

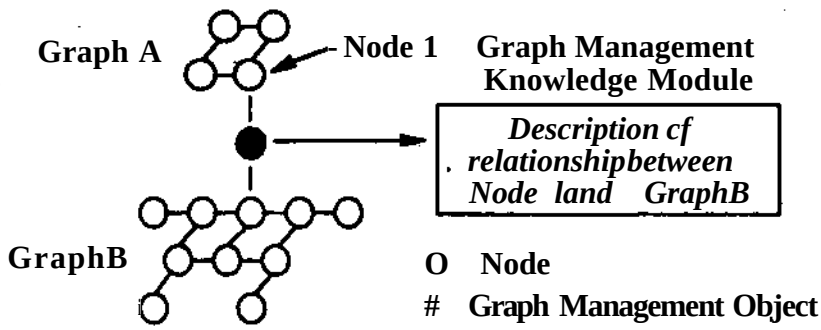

Figure 4. Use of Graph Management Knowledge (Note: Link objects not shown)

Subgraphs are used to help a designer deal with complexity by supporting decomposition and representation at different levels of abstraction. This is done using an *abstraction hierarchy*; where a node in a graph can have a subgraph which is a more detailed representation of that node. Each of the nodes in die subgraph can in turn have subgraphs, and so on. In an abstraction hierarchy, high level (abstract) information is stored at the "top", and more detailed information is stored in subgraphs. The amount of detail increases with depth in the hierarchy.

The buckling column example is too simple to illustrate this idea. Instead, we will use as an example the design of a continuous casting process. In a continuous casting process, molten metal is continuously poured into a mold, withdrawn from the mold, cooled, and undergoes some sort of processing. Figure 5 shows a graph representing a continuous casting process. This is a very abstract, high-level description of the process. Gearly, there is not enough detail here to be able to do any kind of analysis. However, since designing a continuous casting process is a complex task, the initial approach an engineer might take is to decompose the process into subprocesses, in order to reduce the complexity of the problem.

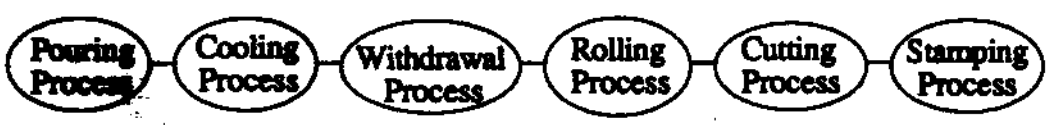

Figure & Graph of Continuous Casting Process (Note: Link objects not shown)

For the design of the entire continuous casting process, each of these nodes would point to a subgraph representing that process in greater detail. Figure 6 shows a possible abstraction hierarchy for this problem (the names of the nodes have been left out). Subgraphs can continue "downward" several levels to capture increasing detail. This is useful if parts of a subgraph are sufficiently complex to waiTant "sub-subgraphs." In general, this concept can be extended downwanls to any level of detail or upwards to an arbitrary level of abstraction. Eastman, *et ai*$_9$ (1991) proposed a similar type of

hierarchy using sets of functional entities. At one level, a functional entity is a single structure, but it may also be composed of other functional entities.

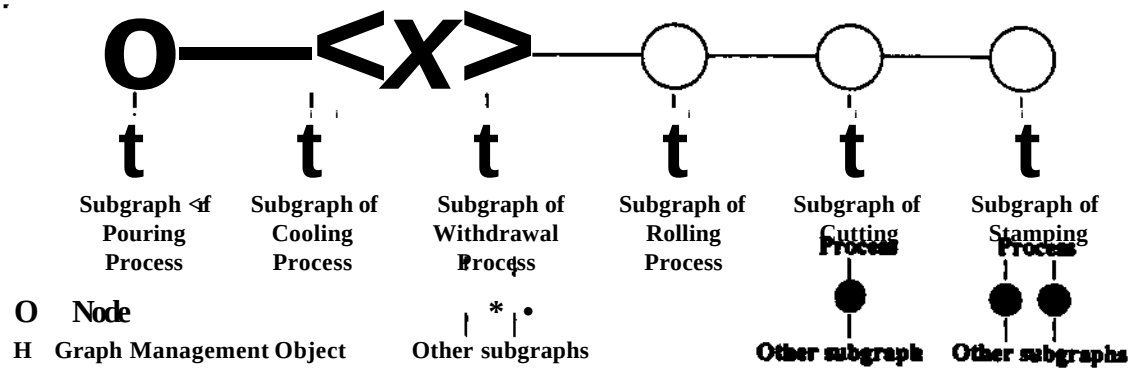

Figure 6. Hypothetical Abstraction Hierarchy (Note: link objects not shown)

Working at an abstract level reduces the amount of information involved with a problem, which is useful when faced with a complex design task. However, the detailed information is necessary at some point (for analysis, for example). Using subgraphs, the engineer can simultaneously represent a system at multiple levels of abstraction. This is done by making the detailed graph a subgraph of a single node, and using that node (an abstraction of the detailed subgraph), as part of the graph of a more complex system. The motivation for including this capability is to allow a designer to capture desired detail, but also when it is convenient, to be able to consider a system or subsystem as a whole.

Figure 5 showed how a continuous casting process could be decomposed. However, as is typical of mechanical design problems, the various subprocesses are not decoupled. If we decide to change the shape of the strip by modifying the mold, which is part of the pouring process, we may need to change the cooling process to accommodate a different molten metal flow. In addition, that change would need to be reflected at the withdrawal and rolling processes as well, since the change in strip shape may prompt changes to those processes as well. By using graph management objects to keep track of the different levels of a design, we provide a means for information flow associated with these types of interactions. Although we are able to approach the design of each subprocess separately, the connectivity between them, and thus the knowledge about potential interactions, is preserved.

3 EXAMPLE: DESIGN OF A ROLLING PROCESS

In section 2, a simple buckling column was used to illustrate the use of the graph representation and its components. The buckling example is rather simple, but it makes several points. First, all knowledge relevant to the problem is efficiently stored in a modular fashion using this representation. Second, topological, domain, and design instance knowledge are maintained independently. Third, a series of different models of interest can rapidly be generated. For instance, changes in material, column geometry or boundary conditions can quickly be made by modifying the values of E, I, L, and/or the boundary condition type in the SSKMs. Columns could easily be added by adding new object-module

tuples to the graph* **The** domain of analysis can be changed by changing pointers to a different DKM These kieas **cany am I* models** of more complex systems and more complicated types of analysis.

In this **secttoflgHE present** a more detailed example which has been implemented using this representation- **if]^poblem** is to design a two-high rolling process, to be used as part of the continuous **casting process** described in the previous section. The design begins at the conceptual level, and the final result is a feasible rolling process design. The emphasis of this section is on the use of the representation and not on the analysis being done, which will be abbreviated The equations used in this section and the assumptions made in their derivations are presented in greater detail in Kalpakjian (1991). The set of specifications which we will be **using** are described in Appendix A. These specifications are presumably imposed due to specifications on the overall manufacturing process, such as the thickness of the parts being made and how many must be produced per **day.**

Figure 7 shows a typical two-high rolling process. In this process, a continuous metal strip having a certain height, enters a pair of rolls which are rotating at a constant angular velocity, and exits at a reduced height

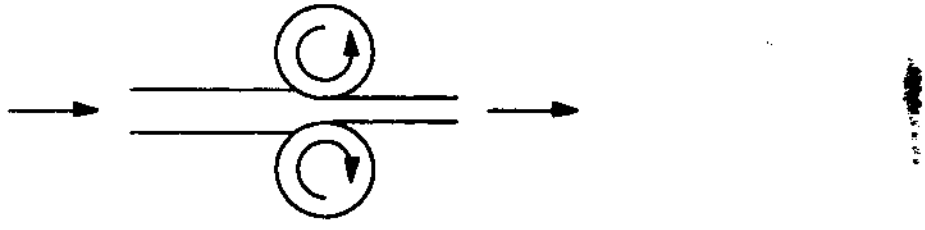

Figure 7. Typical Two-High Rolling Process

Figure 8 shows the graph representing the initial attempt at a design for the rolling process. Four of the nodes and two of the links in the graph point to system-specific knowledge modules which contain information relevant to the design. In this case, the relevant infonnaticm for the strip being rolled is its height (inches), width (inches), and material properties K and n (explained in Appendix A). The relevant infomiation for the rolls is the radius, R (inches), and die angular velocity, co (rpm). Note that the coefficient of friction is not a property of the strip or the roll, but rather the interface between the two, so this information is stored in the SSKMs associated with the links, as shown in the figure.

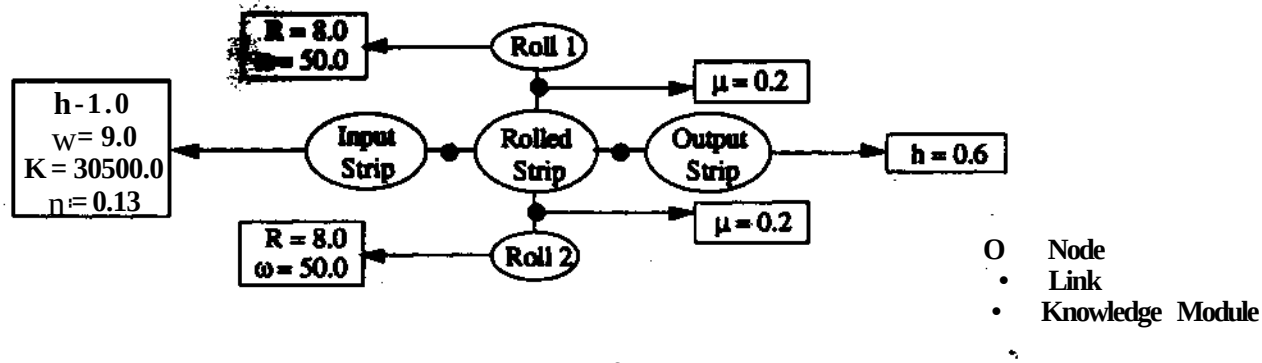

Figure 8. Subgraph of the *Rolling Process* Node

Rolled Strip, the center node, is simply a dummy node used to "connect" the input and output strips to the rolls. The node *Rolled Strip* points to a domain knowledge module called TWO-HIGH-KM, which is not shown in the figure. The DKM contains six equations which describe the physical behavior of a two-high rolling process. These equations, numbered (2) through (7), are contained in Appendix B.

An analysis of the initial design reveals that with the given values of \i and R, Ahmax, the maximum allowable change in height, is 0.32. The specified Ah is 0.4 which exceeds the maximum possible value, so the initial attempt shown in Figure 8 is not feasible. To increase Ahmax, we can increase |i or R. However, increasing friction raises power requirements, as can be seen by equations (2) and (4), so we decide to increase the roll radius. A quick calculation shows that increasing R to 10.0 inches will make the specified Ah possible. The change is made directly by changing the values of R in the SSKMs for *Roll 1* and *Roll 2.*

Now the modified design is analyzed. The required power for the two-high rolling process is 1015 hp. This is more than the available engines are able to supply (500 hp) so the design is still infeasible. Since CD is fixed and \i is already as low as it can be, we decide to add a second rolling station and split the rolling process into two stages, each reducing the strip height by 0.2 inches. Therefore, we can use a second engine for the new stage and still have both rolls in each stage driven by a single engine to get even rotation. To take work hardening into account, the width, w, and the initial height, ho, used for each stage are the *unstrained* values for the strip (which are the values before any rolling took place), giving an upper bound on the power required. Therefore, the values of w and ho used for the analysis of each stage are 9.0 inches and 1.0 inch respectively, but the exiting heights, hf are different.

Another advantage of adding a second stage is that now the Ah through each rolling station is reduced. This allows us to reduce the roll radius and still not exceed the maximum allowable value of Ah, given by equation (7). As can be seen from equations (2), (3) and (4), the reduction in radius will reduce the required power. Since we halved Ah, we halve the roll radius to 5.0 inches.

The graph representing the design is modified by changing the values for R in the system-specific node knowledge modules. Next, four new nodes and links are created, given SSKMs as before, and connected to the existing graph. A pointer is then set from the new node *Rolled Strip 2* to the DKM, TWO-HIGH-KM. The new design is shown in Figure 9 Ginks and knowledge modules are not shown).

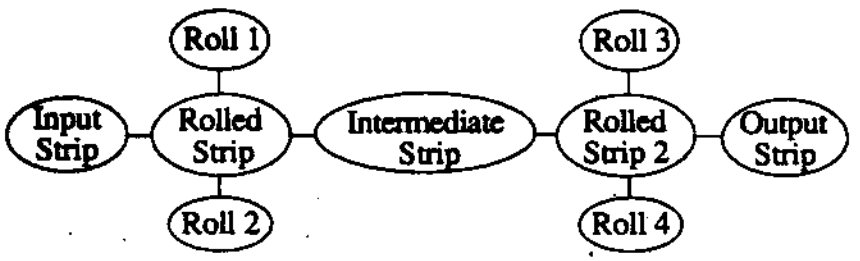

Figure 9. Modified Subgraph of the *Rolling Process* Node

Results of the analysis of this new design are that the first station requires 203 hp and the second station requires 478 hp (the difference is due to strain-hardening). Both values are below the maximum power the engines can deliver, so this design is feasible. We also notice that the total power for the

process is 681 hp* a 33% decrease from the previous value of 1015 hp. Clearly, adding stations and reducing die rolf fldjkiis improves the design considerably. However, adding stations, while reducing operating costs, i||&ases initial investment cost, so there is a tradeoff between initial cost and operating cost wlpfve not considered any economic factors in designing this process.

The use of subgraphs to create an abstraction hierarchy was shown in Figure 6. For this example, we only focused on the design of the rolling process. As described earlier, in practice each part of the continuous casting process would have a subgraph with a more detailed representation. If we were to extend this example to complete the design of the entire continuous casting process, the graph shown in Figure 9 would be the subgraph of the *Rolling Process* node in Figure 6. To model the rolling process, only one additional level of detail is needed However, parts of the subgraphs for the other processes may themselves have more detailed subgraphs if necessary, creating additional levels of depth in the abstraction hierarchy.

Note that from an abstract concept of a continuous casting process, details of a two-stage, two-high rolling process have been developed; the other processes can be similarly implemented This example illustrates the use of alternate abstraction and refinement, generate and test, problem decomposition and coupling, and the interactive process of conceptual design.

4 IMPLEMENTATION

The representation presented in this paper has been implemented using an object-oriented paradigm in Allegro Common Lisp with Flavors, on a Sun workstation. An object-oriented approach is inherently amenable to the creation of "trees" which can represent decompositions for different levels of abstraction (the objects themselves are abstractions of the parts of the design they represent). It also allows components of a design and the connections between them to be generically represented The object-oriented framework lends itself to creating a representation which can be used to model the connectivity between components of a design.

We foresee a mouse-driven user interface to this software, where a designer would be able to rapidly create a design representation, add and delete objects, analyze the designs, and change domains of analysis. The result would be an environment which is easy to use and which supports conceptual design through the process of abstraction.

5 DISCUSSION

This paper addresses issues involved with the use of abstraction in design. We have introduced a knowledge representation to be used as the basis for a CAD tool for conceptual design which incorporates several unique features. One feature is the classification of knowledge shown in Figure 1. The separation of system-specific knowledge and domain-specific knowledge allows a designer to perform different types of analysis on a given design, by changing pointers to different domain knowledge modules, without changing the topological design representation; Conversely, because DKMs contain no knowledge about the system, the equations contained in them are valid for a class of

designs. Since aDKM can be used to analyze a variety of problems, a new one does not need to be created for an analysis if an appropriate one has previously been created. Ultimately, a library of domain knowledge modules will be created, and a designer will select a particular domain for analysis simply by setting pointers to one of them.

The knowledge classification also allows knowledge about the design instance and knowledge about the domain to be changed independently of one another and the given topology. This leads to a simple procedure for creating and modifying designs as well as investigating design sensitivity to parameter changes. Such modifications are important because design is most often an iterative process. As was seen with the rolling example, an initial attempt at a design may be infeasible, or if it is feasible it may require refining.

Another unique feature introduced is graph management knowledge which is used to create the concept of subgraphs. Through the use of subgraphs, an engineer can decompose a problem, as well as create and manage multiple representations of a design at different levels of abstraction. This, in turn, allows a design to be represented at both abstract and detailed levels simultaneously, thereby supporting the process of alternate abstraction and refinement Graph management knowledge can also be used to transfer information associated with interactions between coupled parts of a design which has been decomposed.

This research was partially motivated by the design representation used for the 1PRINCE design automation system (Cagan and Agogino, 1987). Cagan (1991) used a connectivity graph to model rigid connections between regions to support design innovation. 1PRINCE has been applied in the domains of mechanical structures by Cagan and Agogino and chemical reactor networks by Aelion, *et* a/., (1991). Our representation is more general than that presented by Cagan. However, when the 1st PRINCE methodology is applicable, our representation can be used in conjunction with 1st PRINCE by requiring that all connections be rigid, and by limiting connections to three orthogonal directions.

6 FUTURE WORK

There are several ongoing research issues which must be addressed in the context of this representation. One such issue is that of eitor checks. An expectation for intelligent CAD tools is that process of detecting errors be partially automated. These errors may be ambiguities, semantic t r ors or integrity errors which lead to ill-defined designs or to invalid configurations. At present, the burden of detecting errors is left in the hands of the designer. A designer could clearly benefit from having the system recognize impossible configurations or constraint violations. In addition to inconsistencies widiin a graph, another potential source of problems is discrepancies between a graph and its subgraph.

An issue related to error checking is constraint propagation *(e.g.,* Serrano and Gossard, 19881 Constraint propagation and management often becomes an important concern when a problem i> decomposed into subproblems (Brown and Chandrasekaran, 1984; Mahfcr and Fenves, 1985 Steinberg, *et al.,* 1989). However, most mechanical subproblems are highly coupled and constraint

propagation may be complicated. An example of this need occurs when imposing boundary conditions- Presently, where and when to apply boundary conditions is left to the engineer. However, there are times when boundary conditions *must* be applied. In Figure 10, a system is represented by a graph consisting of two nodes, each of which is represented in greater detail by a subgraph. First, a boundary condition on a node (BC-1) must be inherited by its subgraph, as shown. Second, boundary conditions which did not exist in the parent graph, in this case BC-2, must be imposed on the subgraphs to somehow indicate that die[i4]parent" nodes are connected Communication between graphs can be useful for taking into account coupling between different parts of a system. The use of graph management knowledge for flow of information between subgraphs needs to be investigated further.

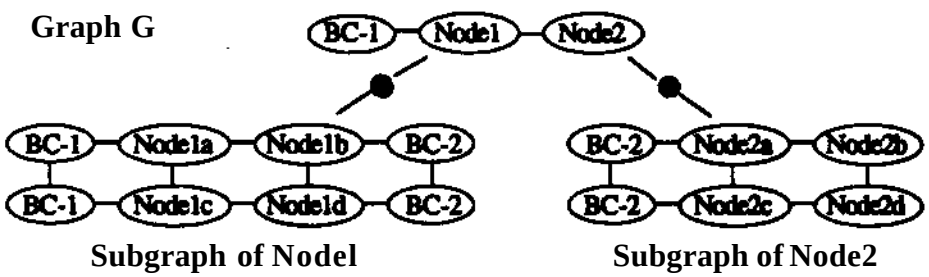

Figure 10. Required Constraint Propagation (Note: links are not shown)

Because design is an iterative and dynamic process, keeping a record of versions of a design would be an important option to make available to a designer, as a design evolves, a designer may choose to keep a trace of design changes and reasons for making those changes. This would make backtracking easier, and could help a designer avoid repeating mistakes. A trace of decisions could be implemented using subgraphs in this representation. The most recent version of a design would point to a subgraph consisting of an earlier version via a graph management object The change made and the reason for modifying the previous design would then be contained in the graph management knowledge module.

7 CONCLUSIONS

This paper presents a knowledge representation which is structured to support design abstraction. Several features for the representation have been motivated, including support of the decomposition of a design problem, analysis of a design, and domain-independent representation of a design. This graph-based representation uses two key concepts to achieve these goals. First, the strict distinction between types of knowledge leads to a domain-independent design representation that allows a design to be analyzed from different perspectives. Modification of graph topology also permits fundamental modification of design concepts. Second, the concept of subgraphs allows a designer to decompose a problem into subproblems and to represent subproblems at various levels of abstraction while still maintaining the inherent coupling between them.

The aim of this research is to produce a knowledge representation whioh allows a designer to computationally create and evaluate new concepts. In supporting problem decomposition and iterative design, and by allowing a designer to visualize concepts at different levels of abstraction, we provide a

powerful tool for use by engineers at the early stages of the design process. Initial applications to engineering **design problems** seem promising. Both the capabilities and limitations of this representation **need** to **be** investigated more fully.

ACKNOWLEDGEMENTS

The authors would like to thank the National Science Foundation for supporting this research under NSF Grant DDM-9108832, and Linda Schmidt for her comments on this manuscript

REFERENCES

Aelion, V., Cagan, J. and Powers, G. (1991), "Inducing Optimally Directed Innovative Designs from Chemical Engineering First Principles," *Computers and Chemical Engineering* (in press).

Brown, D. C. and Chandrasekaran, B. (1984), "Expert Systems for a Class of Mechanical Design **Activity,"** ***Proceedings of the IFIP WG 5.2 Working Conference on Knowledge Engineering in*** *Computer Aided Design,* Budapest, Hungary, September.

Cagan, J. (1991), "A Graph-Based Representation to Support Structural Design Innovation," ***Proceedings of the First International Conference on Artificial Intelligence in Design,*** **Edinburgh,** Scotland, UK, June 25-27, pp 665-682.

Cagan, J. and Agogino, A. M. (1987), "Innovative Design of Mechanical Structures from First **Principles,"** ***AIEDAM: Artificial Intelligence in Engineering, Design, Analysis and Manufacturing,*** 1(3):169-189.

Eastman, C. M., Bond, A. H. and Chase, S. C (1991), "A Data Model for Design Databases," ***Proceedings of the First International Conference on Artificial Intelligence in Design,*** **Edinburgh,** Scotland, UK, June 25-27, pp 339-365.

Gordon, W. J. J. (1961), ***Synectics, the Development of Creative Capacity,*** **Colier Macmillan,** New York, NY.

Hoover, S. P., Rinderle, J. R. and Finger, S. (1991), "Models and Abstractions in Design," ***Proceedings of the International Conference on Engineering Design, ICED 91,*** **Zurich, Switzerland,** August 27-29.

Kalpakjian, S. (1991), ***Manufacturing Processes for Engineering Materials,*** **Addison-Wesley,** Reading, MA, pp 344-362.

Maher, M. L. and Fenvcs, S. J. (1985), "HI-RISE: An Expert System for the Preliminary **Structural Design of High Rise Buildings,"** ***Knowledge Engineering in Computer-Aided Design,*** **J. S.** Gero, ed$_M$ Elsevier Science Publishers, Amsterdam, pp 125-135.

Miller, G. A. **(1956),** [4]Thc Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Review,* 63(2): 81-97.

Newell, A. and Simon, H. A. (1972), *Human Problem Solving*, Prentice Hall, Englewood Cliffs, 1STJ.

Paz-Soldan, J. P. and Rinderle, J. R. (1989), "The Alternate Use of Abstraction and Refinement in Conceptual Mechanical Design," *Proceedings of the ASME Winter Annual Meeting,* American Society of Mechanical Engineers, San Francisco, CA, December 10-15.

Pandurang, P. P., Joskowicz, L. and Addanki, S. (1991), "Context-Dependent Behaviors: A **Preliminary Report,"** ***IntCAD 1991, Preprints of the IFIP WG52 Working Conference on Intelligent*** *CAD,* Columbus, OH, September 30-October 3.

Rinderle, J, R- (1986), "Implications of Function-Form-Fabrication Relations on Design Decomposition **Strategies,"** *Computers in Engineering 1986,*1, American Society of Mechanical Engineers, **July.**

Serrano, **D. and Gossard, D.** (1988), "Constraint Management in MCAE," *Artificial Intelligence in Engineering: **Design, J.*** S. Gero, **e&,** Elsevier Science Publishers, Amsterdam, pp 217-240.

Steinberg, L., Langrana, N. and Fisher, G. (1989), "MEET: Decomposition and Constraint **Propagation in Mechanical Design/*** ***Preprints, NSF Engineering Design Research Conference,*** **Amherst, MA, June** 11-14.

Tomiyama, T. (1989), "Intelligent CAD Systems," *Tutorial Notes, Eurographics '90,* Montreaux, Switzerland, September 2-7.

APPENDIX A Rolling Process Design Specifications

1. The type of raffing process is a two-high rolling process
2. **Material: 5052-O Aluminum (K** = 30500 psi,n = 0.13/
3. Height of input strip (ho): 1.0 inch
4. Height of output strip (hf): 0.6 inches
5. Width of input strip (w): 9.0 inches
6. Angular velocity of the rolls (oo): 50rpm
7. Coefficient of friction *(\i):* 0.2
8. Engines available in-house can deliver a maximum of 500 horsepower
9. Both rolls in a two-high stage must be run by a single engine to insure even rotation
10. Ignore economic considerations

* For a strain-hardening material the stress-strain relation is approximated by **a** = Ke^n, where K is called the strength coefficient and n is called the strain-hardening exponent

APPENDIX B Rolling Process Design Equations

The power (horsepower), P, required is given by:

$$P = 2it(L)(F)(co)/396OOO, \quad (2)$$

where L is the length (inches) of the arc of roll contact and F is the roll force (pounds). L is approximated by:

$$L = [R(ho-h_f)]l/2, \quad (3)$$

where hf **and** ho are the output and input strip heights (inches), respectively, and R is the roll radius (inches). F, **the required** roll force, is given by:

$$F = \mathbf{(LXwXY\text{^}gHl} + nU(ho+hf)], \quad \mathbf{(4)}$$

where w is **the widA** (inches) of the input strip and \i is the coefficient of friction. Y^vg* *[c] average flow stress (psi), is given by:

$$Y^l_{avg}=1.15(K)(en)/(n+l), \quad (5)$$

where e is the true strain, given by:

$$e = \ln(ho/h_f), \quad (6)$$

*

Δh_{max}* the maximum possible change in strip height through a rolling process, is given by:

$$\Delta h_{max} = \mu^2 R. \quad (7)$$