A Theory of Features for Discrete Manufacturing

J.C. Boudreaux
Advanced Technology Program
National Institute of Standards and Technology
U.S. Department of Commerce

Abstract.

Even though the idea of using features to unify for large-scale computer-integrated manufacturing systems has often been proposed, no current system has been able to achieve this goal. This negative result can be explained in part by the fact that what appears to be at the outset a simple matter, turns out to have unanticipated complexity. Features are very hard to get a grip on! There are many lists of features, but nothing has yet emerged which could be plausibly put forward as a full-blown theory of features. In this paper, I will attempt to present a preliminary, rough-cut version of such a theory by outlining what I believe to be its main components. Up to now, most studies of features have assumed that features are chunks of geometry. But it is proposed in this paper that the kernel of features is not geometric, but topological. Features are best represented as undirected planar graphs whose nodes may be assigned expressions that can be subsequently evaluated. Taken collectively, the node evaluation process will then yield suitable geometric interpretations. There are many ways to do this, the most direct of which is to require that geometric interpretation functions map nodes onto points and edges onto curves whose boundaries are the images of the nodes.

1. Introduction.

Information is scattered in repositories throughout the organization: in corporate databases, in data file and application programs, as well as in the memories and accumulated skills of the workforce. This scattering can be explained in part by the history of the adoption of information technology in the organization, but also by the mind-numbing complexity of the information considered in its entirety. Any realistically complete repository would be unimaginably complicated. For practical purposes, a knowledge repository could only contain approximations, and summaries, of this communal store. Workers should be able to use the repository to make sense of the information flows around them. For example, the worker could obtain many different perspectives of a particular product or process, seeing first one aspect stand out as important, and then another.

The skills needed to bring products to market are held by specialists. Product development is a matter of ordering the tasks such that each specialist gets all of the information needed to perform his task, and then to pass the work packet downstream to the next workers in the sequence. The serial character of this description need not imply a strictly sequential, or linear ordering, of the tasks. That is, specialists can pass the intermediate product to several others workers, but further downstream these separate branches will need to be joined together.

2. Organizational Knowledge.

As work packets moved through the organization, the product is progressively developed. At each step, the product consists of bundle of sketches. The point of a sketch is to present that amount of information, general and specific, to compactly evoke chunks of knowledge. Since the primary focus is the resultant behavior, one especially useful way to describe the effort is to code (small) chunks of knowledge in the form of grammatical productions:

```
  <rule>  ::=  ( if <clause> )
<clause>  ::=  ( <guard> ( <action> ) )
 <guard>  ::=  <expression>
<action>  ::=  <expression>
```

These productions say that knowledge may be cast in the form of conditionals, the antecedent clause is a guard expression which, when applied to information flows, is either satisfied or not, and the consequent of which is a sequence of one or more actions. For example, the

following rules indicate that a workpiece is to be machined by a turning operation, then the proposed workholding action depends upon the aspect ratio (width-to-height ratio) of the workpiece:

```
(if
   (and
    (isa machine_assigned(workpiece)
         turning_machine)
    (<= aspect_ratio(workpiece)  0.5))
   (fixture-with face_plate))

(if
   (and
     (isa machine_assigned(workpiece)
          turning_machine)
     (>  aspect_ratio(workpiece) 0.5)
     (<= aspect_ratio(workpiece) 3.0)
   (fixture-with chuck)
```

Each guard must be operationally defined, that is, it must be possible to effectively determine whether it is satisfied in a given situation or not. There are no corresponding restrictions on the actions which may appear in the consequent. For example, one could modify the product sketch itself, or prowl around a library of CNC machine programs for programs used in similar cases, or notify the operator of a sensed error in a pick-and-place positioning system, and so on. It is even possible to construct examples in which the action changes the rules themselves.

3. Structuring Knowledge Repositories.

Workers with specialized knowledge have some view of the repository, but given the complexity of even partial views, it is implausible to imagine that they process the rules randomly. A good indication of a specialist's skill is his ability to filter out the many irrelevant rules and to focus on the remaining few. To do the job efficiently, the specialists must establish a method for ordering the rules so as to avoid unnecessary evaluations of guards, which suggests that the precise structuring of the information repository is sensitively dependent upon the attributes of the product sketch itself. When guards are evaluated with respect to sketches, then the specific attributes of sketches determine whether or not the guards are satisfied. The feature set of a sketch consists of just this aggregation of rule-induced attributes, and thus a feature set is that by which of which an efficient test strategy is possible. This is vague, but it does put us on notice of the deep dependence of features upon the structure of the organization's knowledge repository.

But how should we structure the repository? Or to rephrase the question, what kind of structure should we be prepared to attribute to product (and process) sketches? Based on [Bou91] and [Bou93] as well as [GF92], it is reasonable to suggest the Lisp eval process a theoretical paradigm. Formally, almost any syntactically well-defined language could do the job, but Lisp is especially suited to this task. Lisp is an interpreted, type-free language which relies on one core metaphor: using lists to specify the application of a function (the first, or head, expression of the list) to its arguments (the rest of the list). When combined with the conceptual simplicity of the evaluation model and the expressiveness of the data structuring primitives, the case is even more compelling. The operation of Lisp interpreters is an infinite loop: read an expression from some input source of expressions, called an input stream, apply eval to the expression in the context of some specific symbolic environment, the print the resulting expression to an output stream. The symbolic environment changes dynamically in response to the operation of the interpreter: sometimes only the value of a symbol is changed, but sometimes new symbol-value pairs are added. The richness of the Lisp programming metaphor is perhaps best indicated by the ease with which that the operation of eval itself can be exactly simulated. Since eval is just one function among others, it is possible to nest the primitive eval in a function definition to create many different derived evals, each of which could serve as the basis of a new family of interpreters.

Having selected the framework, the next thing to suggest is that all domain-specific predicates and functions should be interpreted as algorithms, which are in turn modeled as Lisp expressions. The value of all expressions is obtained by evaluating the expressions in a controlled symbolic environment. The following grammar will be instantly familiar to all friends of Lisp:

```
<symbolic_environment>  ::=  ( { <frame> } )
             <frame>    ::=  ( { <symbolic_pair> } )
       <symbolic_pair>  ::=  ( <symbol>  <value> )
```

That is, every symbolic environment consists of a sequence of symbolic frames, each of which is a list of symbol/value pairs.

The representation of sketches has several layers, which differ in scope as well as in degree of abstractness. The most abstract layer associates products with symbolic webs, each member of which has some more-or-less precisely defined meaning and which may be anchored to entities in lower (and less abstract) layers. Webs map straightforwardly into symbolic

environments. This layer contains most of the product representation schemes now being actively studied, including STEP. For example, a **cylindrical_surface** is defined in Express notation as follows:

```
ENTITY
  cylindrical_surface
    SUBTYPE OF (elementary_surface);
    radius : positive_length_measure;
END_ENTITY;
```

which says that all **cylindrical_surface** entities have all of the attributes of **elementary_surface** entities, and that each of these entities is the set of all points at a constant distance, the **radius**, from a straight line. This definition explicitly mentions other symbols, which are defined elsewhere:

```
ENTITY positive_length_measure
   SUBTYPE OF ( length_measure );
WHERE
   WR1 : value_of (SELF, SELF.unit_component) > 0;
END_ENTITY;
```

which says that a **positive_length_measure** is a **length_measure** whose value is greater that zero. The general idea of the first layer is that the joint development of suitable symbolic environments and appropriate evaluation functions can support an abstract product model in which lists of symbols are used to describe very complicated product families.

A second layer consists of combinatorial models in which the sketches themselves are represented by what may be called product decomposition trees: (1) the root node designates the product itself, and (2) every node designates the component which is assembled from the components designated by its immediate descendants. I use "assemble" and "component" in a very broad sense. For example, if a turned part is produced from blank stock, then I will say that the blank is a component of the turned part and that the turned part has been assembled from the blank. If a node has no descendants, then the corresponding component requires no assembly at all. But this remark is a bit misleading. A leaf component is not necessarily unworked. In fact, it can be a complicated assembly in its own right, like an automotive piston. Leaf node components require no assembly by the firm itself, though a great deal of assembly may need to be done by the suppliers.

To see the implications of this structure, we should imagine a tree in which all of the components of the product, including those which are deeply nested, are represented by separate nodes. Each node can be interpreted both as a place in which one or more manufacturing processes are performed on the components (if any) which are passed up by the node's immediate descendants (if any), and also as a source of (partially) finished components. The interweaving of product and process data in a single structure is very useful. In effect, it supplies a single unified support system for many of the control artifacts used in manufacturing. For example, if the tree is read in terms of the processes to be performed, then it contains precisely the information needed to define of the layout of the factories that will be needed to produce the product. If the tree is read as a temporal ordering of processes, then it is the source of the master production schedule which, together with process time standards, controls the flow of production.

Many recent studies have proposed a life-cycle classification of features, such as design features, manufacturing features, machining features, inspection features, and so on. The significance of these proposals may be seen by observing that product decomposition trees are always developed over a long period of time. At first, the tree may consist of a single root node and a bundle of requirements. Over time, and with a lot of effort, new nodes are dropped, each elaborating and refining the earlier stages. At the end, the structure has been fully fleshed out, all of its promissory stubs have been fully expanded, and all of the information and knowledge of the organization has been fully and properly distributed throughout. *Everyone knows that such end states are never actually reached, but the point is that product decomposition trees are rich enough to provide a formally precise, even elegant, definition of product sketches and the processes whereby they are created.*

Finally, the most fundamental role of product decomposition trees is that they form the backdrop, or the framework, with respect to which the featuring of products and their components takes place. The features of any one node may be synthesized or inherited by virtue of its relationship to other (in some cases, all other) nodes. The multiplicity of these relational ties, and the complexity of the procedures needed to unify them, go a long way in explaining the observed diversity of feature categories. Thus, functional features are inherited from above by considering the constraints imposed by the node's predecessors. Mating features are both synthesized from below and inherited from above because of the requirements imposed by the assembly processes. Form features are those through which all of the (often) contending specifications of the part are finally reconciled. Form features are geometric models in terms of which all of the connectivity-induced

information of the product decomposition tree is implemented. The exact structure of this layer may be resolved in several different styles. First, there is the usual geometric level of representations which corresponds to the familiar Brep, constructive solid geometry (CSG), and so on. Second, the part may be represented as a regular set, that is, a bounded, closed, semianalytic Euclidean set, or as a blob-like object in the sense of Koenderink [Koe90]. Both of these methods permit very sophisticated renditions of parts, including the definition and use of parameter-passing mechanisms, but Koenderink's proposal has the advantage of having greater mathematical power.

4. Feature Expressions.

Features have two different, but related, roles. First, features are used *attributively*, that is, they are used to form complex attributes which are applied to external systems. For example, if we have a situated part, that is, a part which has been placed within a specified (physical) frame of reference (consisting of an identified coordinate system and a transformation matrix), then the feature system for that part will support an effective procedure for determining whether or not the part exemplifies the feature system. An analogous step can be put in place for process entities, which can also be framed by feature systems. Second, features can be used *generatively*, that is, they are used as entities or (mathematical) objects in their own right, and are assemble together, or composed, to form more complicated feature systems.

The practical consequences of this insight is that features and designs are far more complicated than has usually be thought and we will have to invent a system of some subtlety to capture these notions. The point is that we need to invent data structures which support design creation, and which, when input to a compiler-like process, will support the elaboration of effective (algorithmically definable) procedures to be applied to concrete cases. The following grammar, which is based on a linguistic framework defined in Chomsky's Government/Binding Theory [Coo88], introduces the notion of a feature expression:

```
<feature_expression>  ::=
      <basic_feature> |
      ( <constructor> ( <feature_expression> ) )
<basic_feature>  ::=
      ( <feature_head>  ( <feature_modifier> ) )
<feature_head>   ::=  <symbol>
```

Feature expressions are either basic or derived by applying a constructor to already derived features. The distinction between these kinds of features is not fundamental. In fact, they different only in their source: basic feature expressions are supplied directly as lexical items, and derived feature expressions have to be constructed by the user. Basic feature expressions consist of a symbol followed by a list of one or more feature-modifier expressions:

```
<feature_modifier>   ::=
      ( <modifier_head> <filter> )
```

In the feature definition the associated expression is a expression which acts as a filter and which is the place which specifies the effect of an instance of the modifier, that is, an occurrence of the modifier followed by an admissible value. An important advantage of this model is that even basic features are constructed to be modifiable, based on the values of a specified list of parameters. Thus, basic features are templates, which, when the modifiers are filled with suitable values, will unambiguously map the feature expression onto an object in the domain of interpretation. By substituting different modifier values, one can expect to obtain a different (but related) object. The collection of all objects so obtained constitute the variation range of the feature.

Feature constructor expressions are functions which allow compound features to be built up out of simpler features. This manner of speaking is intended to suggest an analogy between feature expressions and arithmetic expression: basic feature expressions are numerals, and constructors are the familiar arithmetic functions. But when more accurately expanded, it now seems highly unlikely that the feature domain has such a simple ontology. Arithmetic expressions, however complicated, return at most one number when they are evaluated. And expressions which return the same value, no how much they differ in internal structure, are identical. In contrast, the evaluation of feature expressions is a context-sensitive function of the symbolic environment. That is, the implicit computational model being considered here is one in which feature expressions are collections of symbols that are linked with one another in ways that are foreign to the ways in which numbers are linked. I find it helpful to imagine that feature symbols come equipped with structures very much like tendrils through which needed information is passed about. There are several ways to flesh out this kind of "tendril" linkage. For example, one could claim that all of the feature symbols are bound together according to an object-oriented paradigm, that is, they are either the names of objects, object classes, or

possibly object methods. Or one could claim that the symbols denote agents, connected in a manner reminiscent of Marvin Minsky's society of mind. Both of these approaches are being carefully considered.

5. The Semantics of Feature Expressions.

The semantic domain for interpreting features expressions it is closely related to the one for processes presented in [Bou93]. Feature expressions are to be interpreted as an undirected graphs whose nodes are list values with the following structure:

```
(<node> (<edge_list>) <content>)
```

As usual, a graph is a (possibly empty) set of nodes, each of which is conventionally represented as an integer. The edge list is a (possibly empty) list of all nodes incident upon (edge-connected to) the node in question. The content of a node characterizes its computational potential. Any Lisp expression may be accepted in this position, but obviously some are much more pertinent that others. The final restriction limits features to planar graphs, that is, to those graphs which may be embedded in the plane (or equivalently on the surface of the sphere) in such a way that the vertices are points in the surface, and the edges are curves in the surface which intersect only at their bounding vertices.

The set of functions for operating on graphs include adding a vertex to an existing graph; deleting a vertex from an existing graph and also deleting all incident edges; adding an edge between nodes; and deleting an edge. These functions are constructively complete: all graphs can be constructed from the null graph, that is, the unique graph with no vertices, by suitable applications of these functions.

Using these primitive functions, more complicated functions can be defined. For example, one may **copy** one graph into another by (1) adding one new vertex to the second for each vertex in the first, and then, having defined a bijection from the vertices of the first to the new vertices of the second, (2) edge-connecting the new vertices just in case their images are edge-connected. By recording the new vertices, an **uncopy** function, which undoes the copy action, can be defined as the repeated deletion of the new vertices. It is an easy exercise to prove that, given the bijection, uncopy completely restores the original graph. It is also easy to show that **copy** and **uncopy** preserve planarity.

A much more difficult issue is edge-connecting vertices in the copied graph with vertices of the original graph. For example, once a graph has been copied, then a simple connection can be done by selecting a vertex in the original graph, connecting every vertex connected to the selected vertex with some vertex in the copied graph, and then deleting the selected vertex. This is a variant of (vertex) **cut-and-paste**, and there are others. Given an edge connecting two vertices, we can interpolate a copied graph by connecting each of the original vertices with vertices in the copied graph, and then deleting the original edge. Unlike **copy** and **uncopy**, these operations do not necessarily preserve planarity and thus need to be carefully monitored. Also **cut-and-paste** operations can be completed in many different ways, which suggests the need for an **edit** operation. In this sense, the construction of feature expressions more nearly approximates the composition of an English sentence than the construction and evaluation of an arithmetic expression.

There is a very natural way to use this technical methodology to describe the variation-within-limits of features, that is, the ability of features to form families. To see how, recall the notion of graph isomorphism: graphs are isomorphic just in case there is a bijection from the nodes of one onto the nodes of the other such that two nodes are connected in the one graph if and only if their images are connected in the other graph. In a sense, isomorphic graphs have the same structure, but they need not have the same nodes. Features are structurally identical just in case they are isomorphic as graphs.

Because the content of feature graphs are not included in this definition, structural identity is not especially useful. But it is a simple matter to show that structural identity is an equivalence relation on feature graphs, and thus induces a partition. If we agree to associate symbolic names with feature graphs, it would perhaps be reasonable to actually associate them with these partitions. In effect, this means that we are willing to regard renumbering as identity preserving. This observation suggests a way to parametrize feature graphs: (1) a node will be called a variable node if and only if its content is a variable symbol; (2) any node in that feature whose content is identical to that symbol is a re-occurrence of that symbol; and (3) a substitution instance of such a feature graph is obtained by replacing all occurrences of the symbol in the feature graph with admissible feature graphs. In order to be admissible as a substituent of a parameter the feature graph as a whole must connect to the same nodes as the original node which is being replaced and all of the nodes must be

foreign to those in the original feature graph. Again this is only approximately correct, in fact, it ignores all of the issues pertaining to agreement in content, but it will do for a start.

Perhaps I should say more about content. I can imagine a situation in which the content can contain agent-like directives to modify the feature graph itself. I can also imagine a situation in which the content will include conditional directives of the form "if so-and-so, then do X", where X is some action that could cause not only a change in the local and global structure of the feature graph but also effect the general environment in terms of which the evaluation of the feature graph is taking place. I can even imagine that the environment will contains communication channels so that the evaluation of the content of one node can cause a message to be broadcast to other nodes in the same or in a different feature which have access to the same channel.

What the restriction of features to planar graphs does is to ensure that every feature can be embedded into the surface of a sphere. One this embedding has been done, it is a short step to a full geometric representation as required by this model. But first, it should be noted that once a feature graph has been fixed, there are many possible embedding functions which can get the job done. Imagine the feature graph as a net with very elastic edges. Then to get an idea of the variety of possible embeddings, all one needs to do is to imagine sliding the elastic net over the surface, of the sphere. So long as the net is not torn, the topological structure remains invariant under these transformations. Fix the position of the net, then if we remove the points on the surface under the vertices and all of the edges, the remaining points fall into connected components call faces. Faces are bound by edges, more precisely by sets of edges which can be ordered to form cycles, that is, a sequence of edges immediately connecting a series of vertices such that no vertex, except the first, occurs more than once, and the first and the last vertex are identical. In effect, this construction is the first stage in the elaboration of the final part. *What happens next, under the direct control of the content of all of the nodes of the feature graph, evaluated in a reasonable sequence, within a properly configured symbolic environment, is that this initial stage hardens, or stiffens, into a fully formed geometric model.*

6. Feature Refinement.

What I now propose to do is to identify a very large collection of entities, ranging from content-free feature graphs on the one hand (feature graphs, the content of whose nodes are uniformly nil), to finished manufactured parts on the other, and the intermediate cases being obtained by progressively adding more and more information. In fact, it is the quantity of information which forms the clear basis for this scheme. The initial feature graphs contain very little information, the manufactured parts contain an inexpressibly infinite amount of information, and all of the intermediate cases have some intermediate amount. This intuitive notion can be made precise by the notion of refinement:

One feature refines another feature just in case the second is structurally identical to a subgraph of the first, and the consequences of the content of any node in the second are a subset of the consequences of the content of the image of that node in the first.

This is only part of the correct definition. We also need to clauses which characterize the relationship between the respective environments, but even the present proposal is more than complicated enough to give us a lot to consider. I will be looking more closely at other possible formulations of refinement which have a different inferential basis. For now, let me put forward some of the immediate implications of this discussion which might have some interest in their own right. First, we can completely collapse the distinction between parts and features. We might say that parts are features which are more or less complete, i.e., have by some measure a lot of information associated with them, and features are parts which have more or less scant information associated with them. Though any hard and fast rule here is out of the question, we might say that the resultant of the geometric interpretation is more part-like than feature-like, and so on. Second, refinement has the Church-Rosser property: for any part X, if Y and Z are both refinements of X, then there is a part W such that W refines both Y and Z. Third, manufactured parts have a Strong Interpolation property: if M is any manufactured part and X is any part such that M refines X, then there is a part Z, different from X, such that M refines Z and Z refines X.

7. Conclusion

The results reported here are an outline of an architecture for a practical theory of features for discrete manufacturing. An enormous amount of technical work, and an extensive prototyping effort, will have to be carried out in order to determine whether this programmatic sketch has any real power to illuminate this obscure and difficult area. For whatever progress as has been made, it is my pleasure to acknowledge the intellectual debt that I owe to my colleagues Scott Staley of Ford Research and Richard Crawford of the University of Texas at Austin. The general thrust of the position described in this report is the result of many vigorous, and occasionally intense, technical discussions, beginning in 1992 and continuing even now. Such errors and infelicities that remain in this work are entirely my own responsibility.

Bibliography

[AS85] Abelson, H. and Sussman, G.J. *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[AO91] Apt, K.R. and Olderog, E.-R. *Verification of Sequential and Concurrent Programs*, Springer-Verlag, New York, 1991.

[BP83] Barwise, J. and Perry, J. *Situations and Attitudes*, MIT Press, 1983.

[Bou87] Boudreaux, J.C. "Computational Ontology," in J.C. Boudreaux, B. W. Hamill, and R. Jernigan (eds.), *The Role of Language in Problem Solving - 2*, North Holland, Amsterdam, 1987; 133-160.

[Bou91] Boudreaux, J.C. "Code generation techniques to support device abstraction," *Proc. International Conference on Manufacturing Systems and Standardization*, Budapest, Hungary, 1991; 1-9.

[Bou93] Boudreaux, J.C. "Concurrency, Device Abstraction, and Real-Time Processing in AMPLE," in W.A. Gruver and J.C. Boudreaux (eds.), *Intelligent Manufacturing: Programming Environments for CIM*, Advanced Manufacturing Series, Springer-Verlag, 1993, 31-91.

[Coo88] Cook, V.J. *Chomsky's Universal Grammar: An Introduction*, Basil Blackwell, 1988.

[Dij76] Dijkstra, E.W. *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Gel91] Gelernter, D. *Mirror Worlds*, Oxford University Press, 1991.

[GF92] Genesereth, M.R. and Fikes, R.E. "Knowledge interchange format, version 3.0 reference manual," Technical Report Logic-92-01, Stanford University Logic Group, 1992.

[Gru92] Gruber, T.R. "Ontolingua: A mechanism to support portable ontologies," Technical Report KSL 91-66, Stanford University, Knowledge Systems Laboratory, 1992.

[GL91] Guha, R.V. and Lenat, D.B. "CYC: A Mid-Term Report," *Applied Artificial Intelligence*, vol 5(1991), 45-86.

[Koe90] Koenderink, I. *Solid Shape*, MIT Press, Cambridge, MA, 1990

[LM90] Lee, J. and Malone, T.W. "Partially Shared Views: A Scheme for Communicating among Groups that Use Different Type Hierarchies," *ACM Transactions on Information Systems*, vol 8(1990), 1-26.

[NFF91] Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., and Swartout, W.R. "Enabling Technology for Knowledge Sharing," *AI Magazine*, vol 12(1991), 36-56.

[PT91] Pan, J.Y.C. and Tenenbaum, J.M. "An Intelligent Agent Framework for Enterprise Integration," *IEEE Transactions on Systems, Man, and Cybernetics*, vol 21(1991), 1391-1407.

[Pet92] Petrie, C.J. (ed.) *Enterprise Integration Modeling: Proceedings of the First International Conference*, MIT Press, 1992.

[SR90] Stankovic, J.A. and Ramamritham, K. "What is Predictability for Real-Time Systems," *Real-Time Systems*, vol 2 (1990); 247-254.

[Qu92] Quinn, J.B. *Intelligent Enterprise: A Knowledge*

and Service Based Paradigm for Industry, Free Press, 1992.

[Wea88] Weatherall, A. *Computer-Integrated Manufacturing: From fundamentals to implementation*, Butterworths, Boston, 1988.