POSITIVE-NEGATIVE FEATURE INTERACTIONS IN COMPUTER AIDED PROCESS PLANNING

Thesis

Submitted to

Graduate Engineering & Research
School of Engineering

UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for

The Degree

Master of Science in Mechanical Engineering

by

Robert A. Dietrick II

UNIVERSITY OF DAYTON

Dayton, Ohio

April 1992

POSITIVE-NEGATIVE FEATURE INTERACTIONS IN COMPUTER AIDED PROCESS PLANNING

APPROVED BY:

John P. Eimermacher, Ph.D.
Advisory Committee, Chairman
Professor, Mechanical and Aerospace Engineering Department

Franklin E. Eastep, Ph.D.
Interim Associate Dean/Director
Graduate Engineering & Research
School of Engineering

Patrick J. Sweeney, Ph.D.
Interim Dean
School of Engineering

ABSTRACT

POSITIVE-NEGATIVE FEATURE INTERACTIONS IN COMPUTER AIDED PROCESS PLANNING

Name: Robert A. Dietrick
University of Dayton

Advisor: Dr. John P. Eimermacher

The Rapid Design System (RDS), a United States Air Force sponsored research project, is an object-oriented system composed of feature-based design, fabrication, and inspection sub-systems. The RDS permits a user to design a part using features and to then automatically generate the process plan including the Numerical Control (NC) code to machine the part.

In the design sub-system, the user may select from both positive (i.e. a rib feature) and negative (i.e. a pocket feature) features to create a part. The research addressed by this paper concerns the special issues associated with positive features in the fabrication sub-system. Analysis and classification is performed on positive features relative to process planning issues. In particular, a comprehensive scheme for addressing the case of pocket-island interactions is presented in detail. The primary issues of these interactions are tool selection and NC boundary definition for the negative volume associated with one or more positive features. The negative volume is mapped into a set of

intersecting sub-features. After performing generative machining process planning on these sub-features to determine tooling and speeds and feeds, NC boundaries are defined.

ACKNOWLEDGEMENTS

I would like to express my appreciation to Dr. Fred Bogner, Dr. Ron Deep, and Dr. John Eimermacher for serving on my thesis committee. Also I would like to thank David Domermuth for countless and very productive discussions specifically concerning this research and Dr. Steven LeClair for all of his comments about this and other papers. Most of all, I would like to thank my fiance, Kelly Cassell, for all of her love and inspiration.

VITA

October 30, 1967	Born: Linthicum Heights, MD
December, 1990	B.M.E., University of Dayton, Dayton, OH
April, 1992	M.S., University of Dayton, Dayton, OH

FIELDS OF STUDY

Major Field:	Integrated Manufacturing, University of Dayton

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Over the past several years, companies have begun to realize that conventional Computer Aided Design (CAD) systems using two dimensional primitives (lines, arcs, circles, etc.) in either a two or three dimensional workspace, are not capable of being effectively integrated with manufacturing and business software to form an integrated company computer architecture. One of the principle shortcomings of these conventional CAD systems is their inability to capture critical information such as design intent, geometric relationships, material selection criteria, and manufacturing rules that impact design.

In response, object-oriented systems are being developed as the next generation of CAD. By object oriented, it is meant that the computer language (i.e. Lisp) permits the creation of "objects." These objects are advanced data structures that allow large amounts of information to be associated with them in the form of properties.

One such research project is the Rapid Design System (RDS), a United States Air Force sponsored project. The objective of the RDS project is to develop a software package that will enable the user to design a machinable part and

automatically generate the manufacturing and inspection process plans. Furthermore, by combining the design, fabrication, and inspection sub-systems with a special memory (the Episoidal Associative Memory or EAM) that has the ability to "learn" by experience, good and bad practices, the user will be provided with a knowledge base to assist the design, fabrication, and inspection processes (See Figure 1).

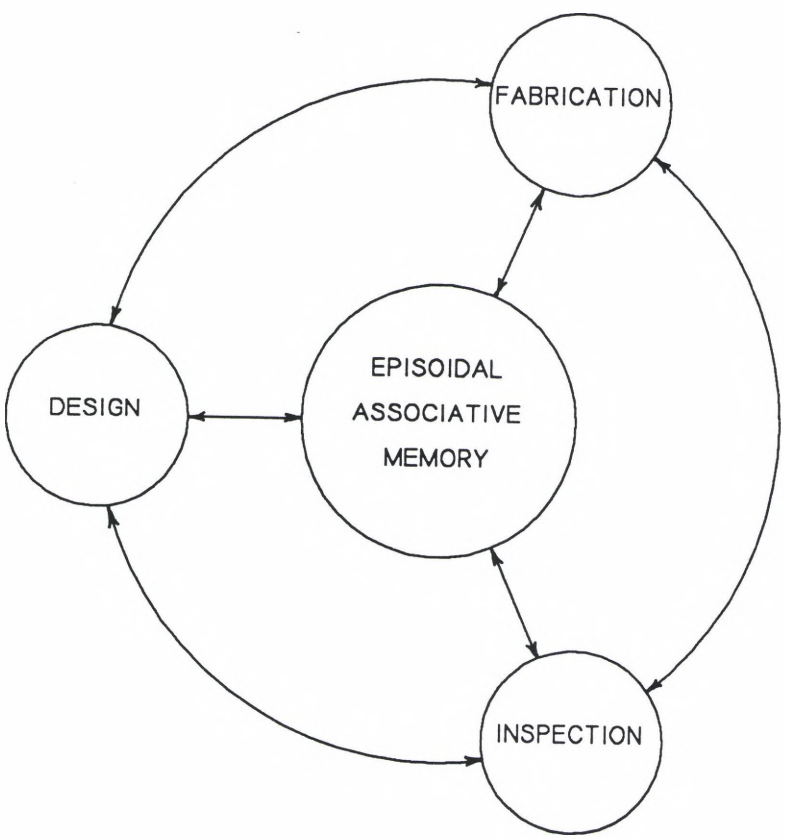


Figure 1: The Architecture of the RDS

One of the several universities involved in the project is the University of Dayton which is responsible for the fabrication sub-system. The role of the fabrication sub-system is to develop the complete machining operations process plan for a given part. This includes determining which machine will be used to produce the part, the necessary machining operations, tool selection, speeds and feeds, and the generation of the NC code to drive the machine.

To accomplish this, the fabrication sub-system is built around MetCAPP, a generative process planner developed by the Institute of Advanced Manufacturing Sciences, Inc., Cincinnati, Ohio. To provide MetCAPP with the proper information, there is a feature translation module that translates design features to manufacturing features. After MetCAPP is consulted to determine the required tooling, speeds and feeds, and pass logic for each feature, the operations sequencing module organizes the operations into an efficient process plan. With all of this information, the NC generation module automatically generates the NC code for the part. Tying everything together and enabling the user to modify the results of any step, is the user interface module. The research addressed within this paper is an expansion of the feature translation module.

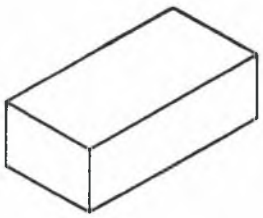
Design Philosophy

In the RDS, designing is accomplished by the use of features. These features are basically three dimensional "building blocks." Primarily, there are two types of features that a designer can use. They are positive and negative features. As the names imply, positive features represent physical material or positive volume whereas negative features represent an absence of physical material or negative volume. Figure 2 describes some of the design features available in the RDS.

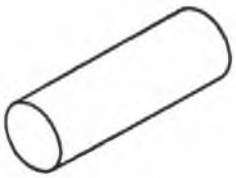
Although a designer could design or "build" anything through the exclusive use of either all positive features or alternatively, all negative features and some positive starting block, the RDS has incorporated both positive and negative features to enhance the system. Primarily, there are three basic advantages of incorporating positive design features:

1. to provide the designer with a more flexible environment,
2. to reduce the amount of time required to design a given part, and
3. to help further define design intent.

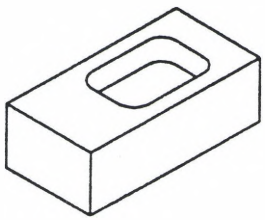
The first two advantages listed above have more to do with versatility than anything else. The third advantage, however, is significantly more important. This is because two designs may be geometrically similar, but still have vastly different purposes.



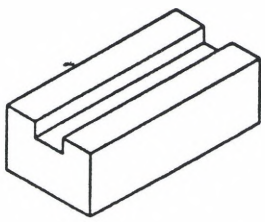
Prismatic starting block



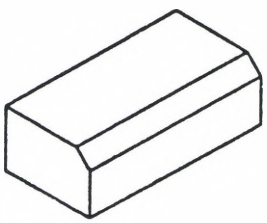
Cylindrical starting block



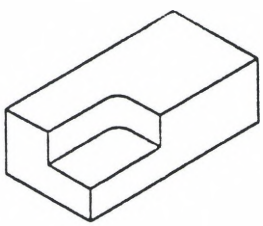
Pocket



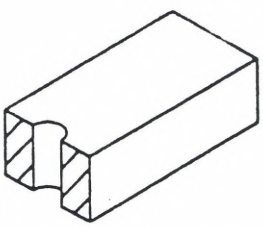
Through slot



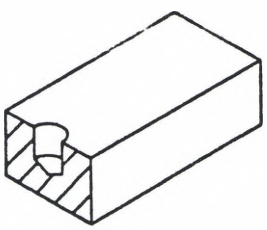
Edge cut



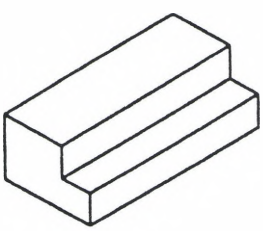
Corner step



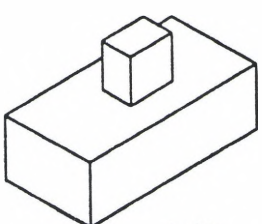
Through hole



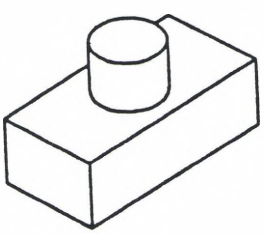
Blind hole



Open step



Rib



Boss

Figure 2: Some Design Features of the RDS

Consider the example in Figure 3 which illustrates an advantage of providing both positive and negative features. In part A, the starting block is a plate. By adding a rib feature to this plate it is implied that the plate thickness is desirable, but the strength or stiffness of the plate is not adequate due to the applied loads. In part B, the starting block is rectangular bar stock. By adding two shoulder cuts to this bar stock, it is implied that the bar stock height is preferred but that the weight or space/fit requirements necessitate the removal of material. So by including positive and negative features, it is possible to more accurately represent the intentions of the designer and as was previously mentioned, this is a primary reason for developing object-oriented CAD systems.

The Challenge of Positive Features

The philosophy of the object oriented RDS is that each feature should be capable of "making itself." That is each feature should be able to determine the necessary tooling, speeds and feeds, and NC path required to physically produce that feature. To assist this process, the RDS incorporates the software package MetCAPP, a generative process planner.

Relative to negative features this involves a reasonably straightforward, though not trivial approach. This is because a negative feature represents in and of itself the material to be removed by a machining process. For example, a 7"L x

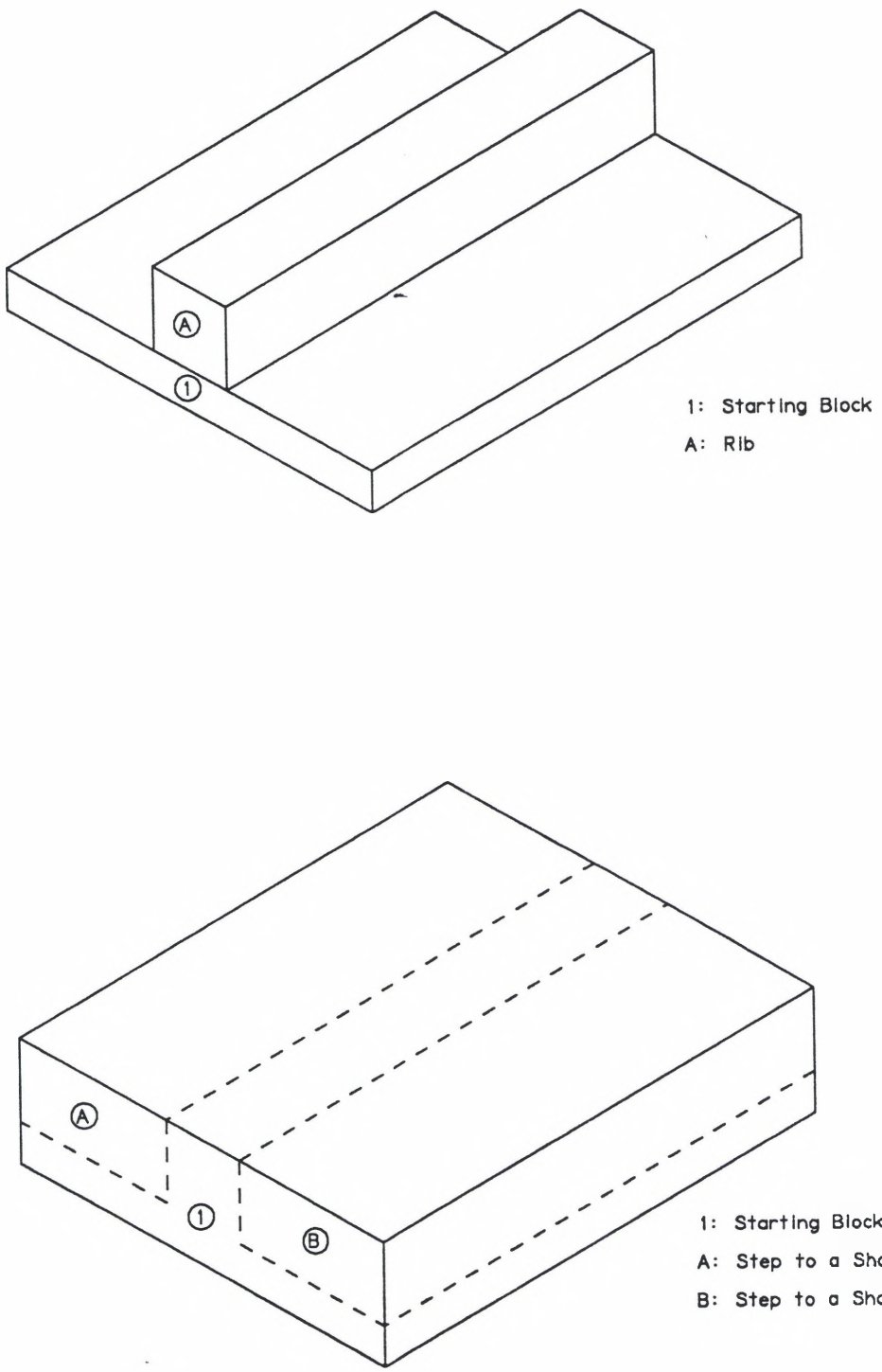
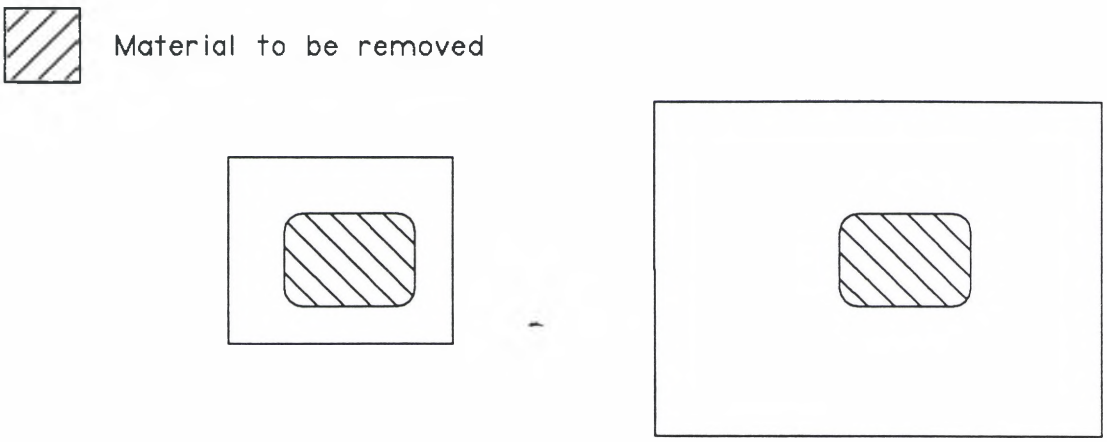


Figure 3: Designing with Positive and Negative Features

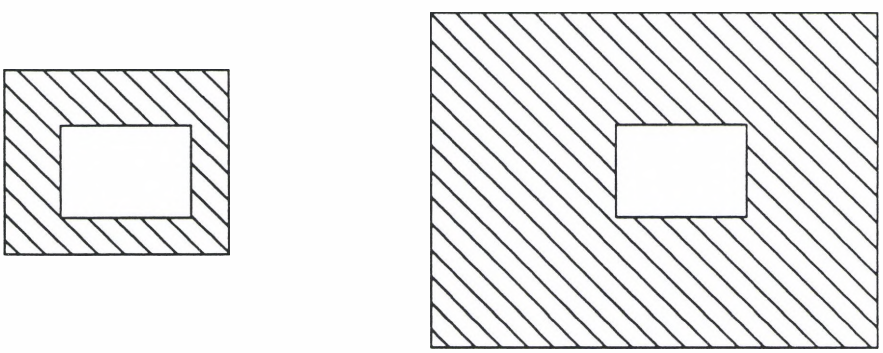
4"W x 10"D pocket will require the same tooling and the same NC path independent of whether it is on a 10"L x 10"W x 10"D starting block or a 30"L x 30"W x 10"D starting block provided that all other things remain constant.

Conversely, a positive feature, such as a rib feature, of the same dimensions could potentially require two entirely different sets of operations to produce itself depending on the size of the starting block or the size of the pocket if the rib is an island in a pocket. At the very least, the NC path must be different because of the different amounts of material to be removed (See Figure 4). This is a result of the inability of a positive feature to inherently communicate the necessary information to produce itself. The reason for this, is that a positive feature is produced by altering the set of operations required to produce some corresponding negative feature, thus leaving behind the positive feature. Figure 5 provides further explanation. In this figure, two rib features, A and B, have been placed in a pocket feature, 1. The pocket feature contains within itself all of the information that is required to select appropriate tooling and determine the NC path. In short, the function f that determines tooling and NC path is a function of the following variables:

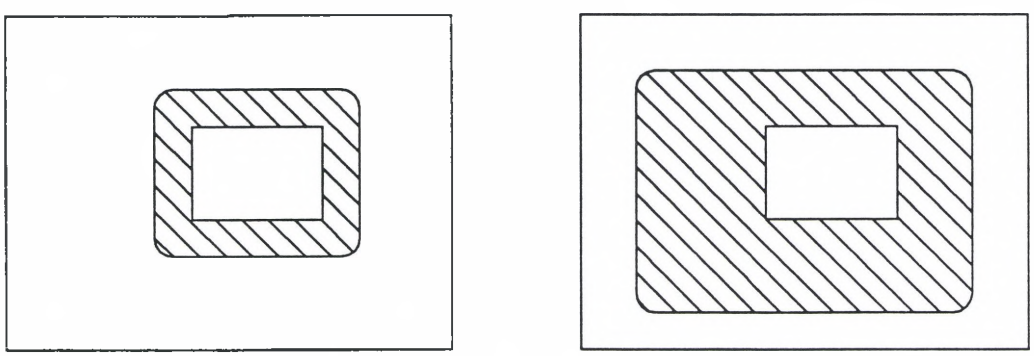
1. type of feature (description of negative volume),
2. length of the volume to be removed,
3. width of the volume to be removed,
4. depth or height of the volume to be removed,
5. center (x, y, z) of the negative volume,



a) Pockets on Starting Block



b) Ribs on Starting Block



c) Islands in Pockets

Figure 4: Material to be Removed

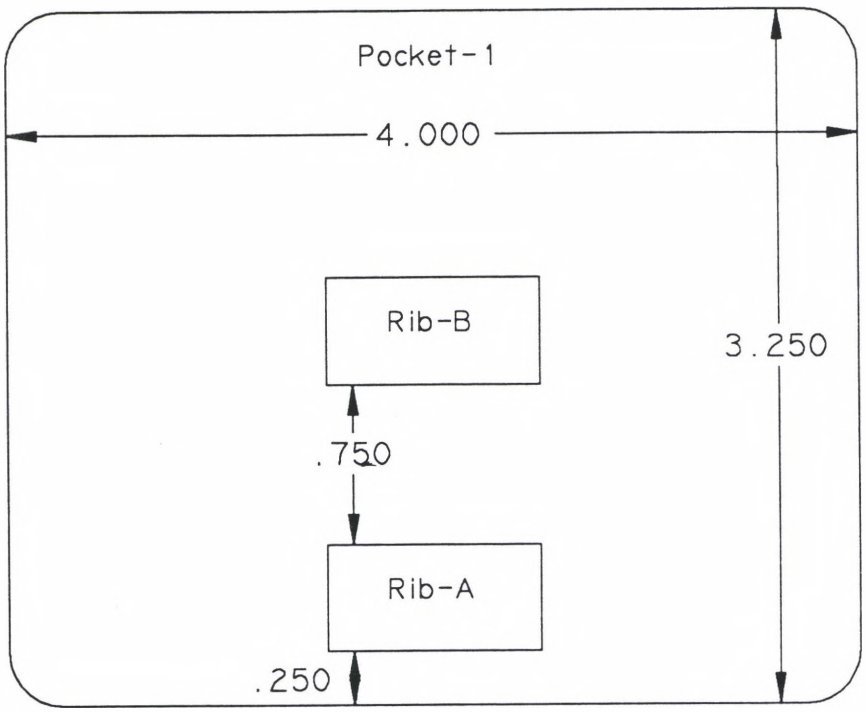


Figure 5: Islands in a Pocket

6. corner radius, and
7. fillet radius.

A positive feature provides the length, width, height, and center location of the volume to remain, not of the volume to be removed. In particular, the location of the rib features relative to each other and relative to the location of the pocket feature can have a profound impact on the tooling required to efficiently produce the resulting geometry. Note that the position of rib A relative to pocket 1 and rib B will require a tool with a diameter not greater than 0.25 inches. It is probably undesirable to use such a small tool, however, to machine the entire geometry. This results in a need for a more complex tool selection scheme.

Figure 5 should also illustrate the need to modify the corresponding negative feature (in this case pocket 1 for both rib A and rib B) as opposed to attempting to capture the length, width, height, and center for function f in the positive features. As can be seen, the information required to generate the operations to produce rib A involves features 1 and A. Similarly, the information required to generate the operations to produce rib B involves features 1 and B.

The information required to generate the operations to produce both rib A and rib B, however, involves features 1, A, and B. Therefore, it is more convenient to view features A and B as impacting feature 1, than to view features 1 and B as impacting feature A and features 1 and A as impacting feature B (See Figure 6).

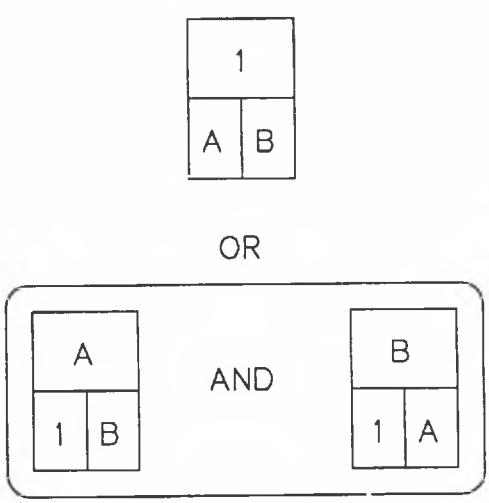


Figure 6: Relationships between Interacting Features

Problem Statement

Positive features present a two-fold problem to the fabrication sub-system of the RDS. First, positive features must be recognized to the extent that it is necessary to determine the corresponding negative features that they impact. In the case of a positive feature that exists on a surface of the starting block, there is no distinct negative feature that is affected. In the case of a positive feature on the surface of the starting block, a corresponding negative feature, such as an open flat rectangular surface, could be created.

The second part of the problem, is the need to alter the set of operations associated with the corresponding negative feature such that the positive feature will be produced. This involves tool selection and NC boundary definition for the resulting negative volume. In this context, tool selection also implies a degree of tooling optimization. Due to the relative ease of generating NC code for a rectangular region of removal, rectangular regions are preferred for the defined boundaries.

The Scope of Investigation

The scope of investigation is defined in two ways. First, it is necessary to define the final form of output from the positive features module of the fabrication sub-system. This final form should include tooling information, vertices,

and tool entry and exit information for each of the regions to be machined. Tooling information refers to the tool diameter, length, material number of flutes, and other information required to uniquely define each tool to be used. To efficiently handle all of this data, each required tool is represented by its unique MetCAPP identification code. The information provided to the NC generator by the positive features module must be sufficient to generate the NC code.

Second, a decision must be made about how many different kinds of positive-negative feature interactions should be addressed. There are currently only two types of positive features--rib features and boss features. Furthermore, because rib features allow the user to define a corner radius, a boss feature is nothing more than a rib feature with equal width and depth dimensions and a corner radius equal to one half of the width.

Having limited the number of positive features to one, there are a total of eight positive-negative feature interactions. This number is arrived at because the rib feature can interact with any of the negative design features or a surface of the starting block. Given the negative feature involved in an interaction, the characteristics of the rib features involved will characterize the specific class of interaction (See Figure 7).

The scope of investigation for this paper will be limited to aligned islands in rectangular pockets. Part of the

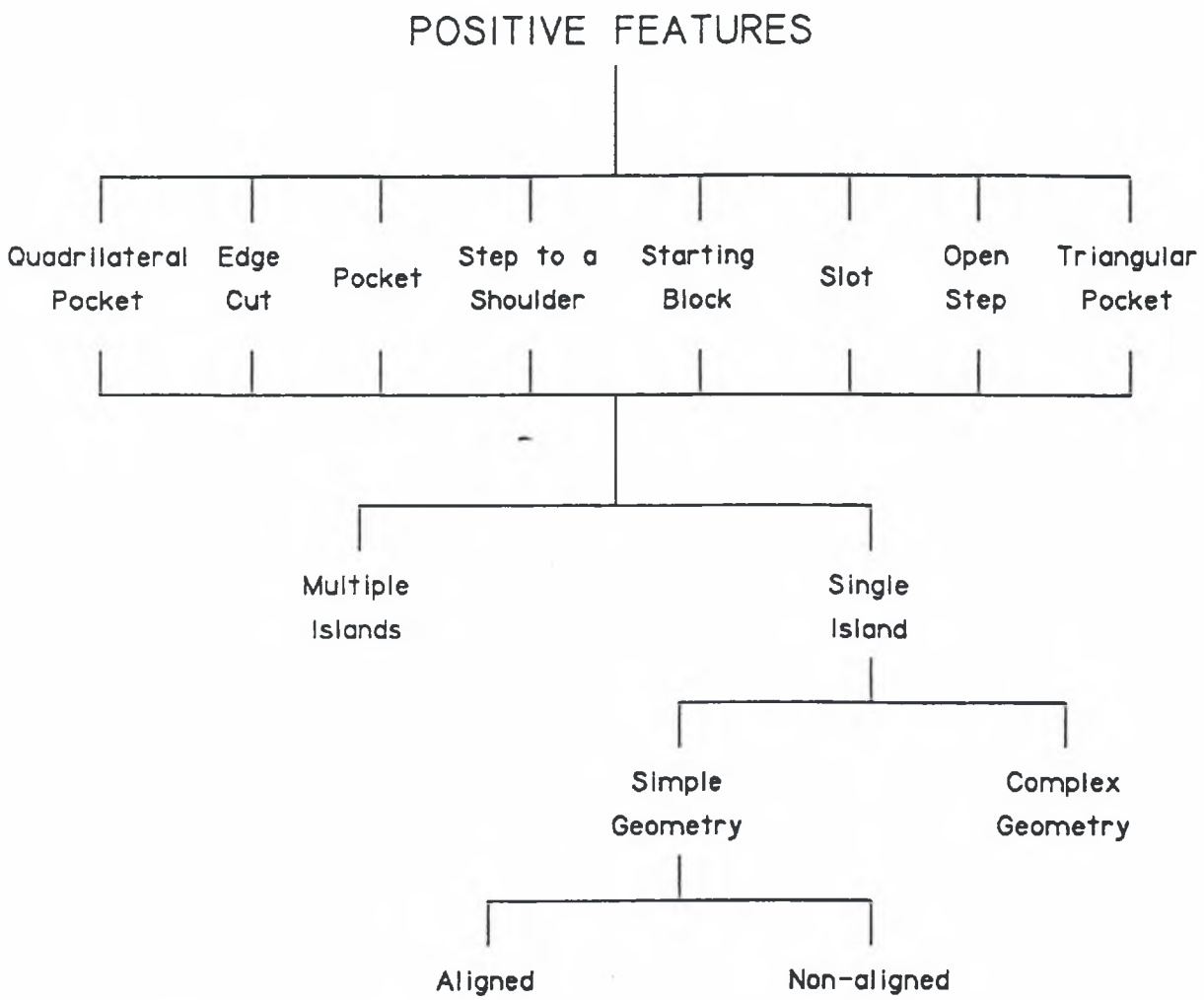


Figure 7: Positive-Negative Feature Interactions

rationale behind this scope, is that virtually all of the negative features currently available in the design sub-system of the RDS can be represented by rectangular pockets. Another reason for selecting pocket-island interactions is to avoid any fixturing interferences. A positive feature in a fully enclosed pocket will not impact the fixturing requirements for the part. This prevents any further complication relative to the need to avoid collisions between the cutting tool or

machine spindle and the fixturing devices. Once the cutting tool is within the pocket area, it is free from possible collisions. In addition to this, the NC rules for machining a rectangular pocket are relatively simple and well defined.

As the RDS is used by the machine shop of the 4950th Test Wing, the feature translation module will have to be further refined. The next logical expansion should be to include positive features that exist on a surface of the starting block. With a relatively minimal amount of new code, this should be easily accomplished.

CHAPTER II

ANALYSIS OF THE PROBLEM

With an explicit problem statement and a well defined scope of investigation, the logical approach is to first establish a means for recognizing rib-pocket interactions and to then process these interactions to the desired extent. Since any design pocket feature could potentially be a corresponding negative feature for one or more design rib features, the positive feature recognition will examine each of the design pocket features.

It is important to realize that even a single island may be composed of multiple rib features (See Figure 8). The importance of this fact, is that any given rib, such as rib B in the example, may impact only a section of a pocket without being attached to the bottom of the pocket. This leads to the concept of viewing and analyzing a positive-negative feature interaction in terms of layers. Each layer of an interaction must have a constant geometry with respect to the height or y-direction except for the fillet radius of either the positive or negative features interacting.

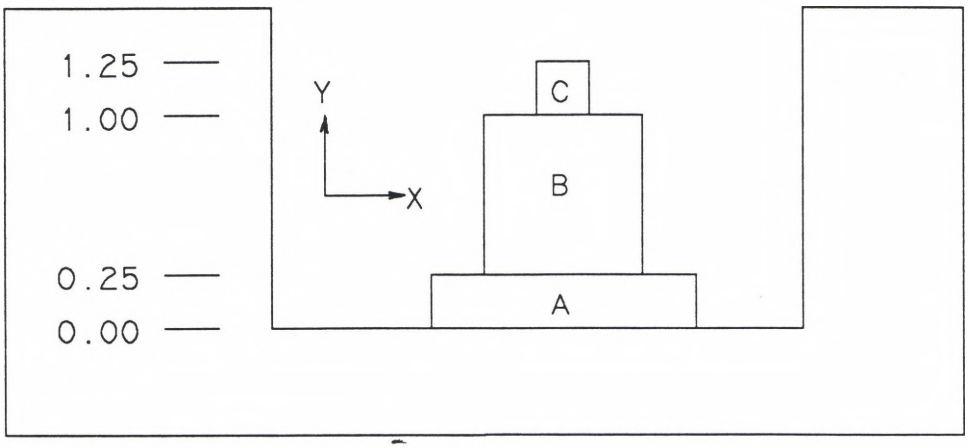


Figure 8: Multiple Ribs Composing a Single Island

In Figure 9, an example interaction involving three islands composed of a total of six ribs is decomposed into six layers. Each of these six layers may now be processed independently with the NC code being generated separately for each layer.

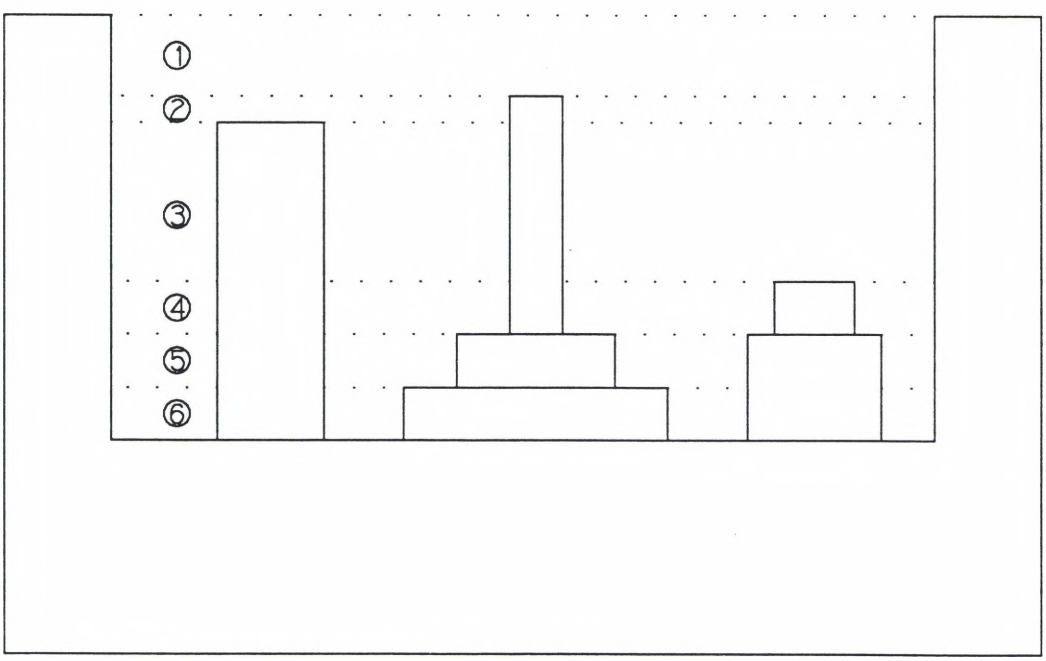


Figure 9: Height-Layers of an Interaction

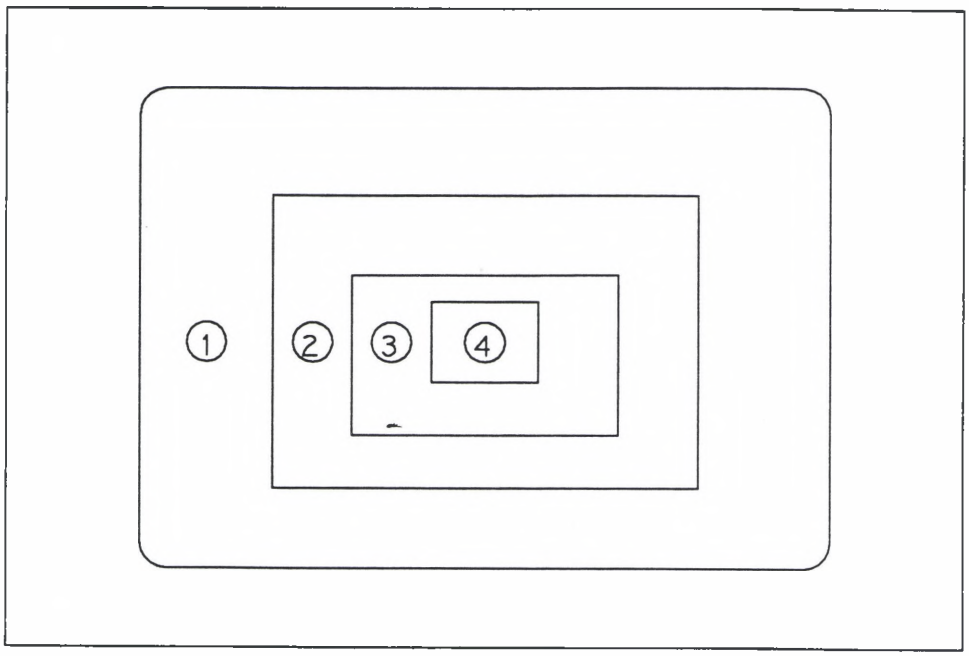
An alternative to the previously mentioned height-layer is the width-layer. The width-layer is formed by a constant geometry with respect to the width or x-direction. For an illustration of the width-layer, please refer to Figure 10.

The principle advantage of using width layers, is that in some cases, consecutive height layers could be machined in a single pass but because the NC code is generated separately for each layer, they will be machined separately which is less efficient. Overall, however, the marginal benefits gained by utilizing width-layers is insufficient to offset the increased complexity of these layers.

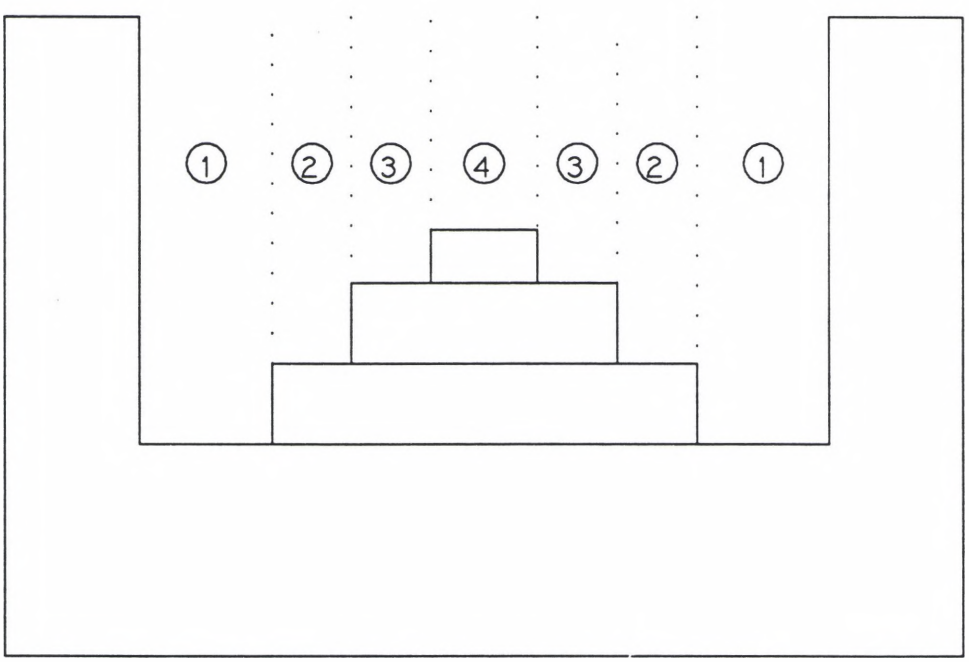
Single Islands

Having briefly addressed the issue of positive feature recognition, the processing of positive-negative feature interactions can now be addressed. Starting at a simple level, the analysis begins with a single island in a pocket. It is important to remember that each height-layer is analyzed and processed independently. As a result, the figures in the following sections are top views instead of cross-sections.

To investigate different techniques of altering the set of operations required to produce the corresponding negative feature, it is necessary to decompose the problem into its two sub-problems. As was previously mentioned, these are tool selection and NC boundary definition.



(a) Top View



(b) Cross-Section

Figure 10: Width-Layer Alternative

The Translation Process

In order to understand how pocket-island interactions are processed, one must first understand the translation of design features into manufacturing features. Currently, each design negative feature is translated to a manufacturing negative feature on a one to one basis (see Table 1). The resulting manufacturing feature can be broken down into the following three features: the geometry feature, the MetCAPP feature, and the NC features.

Table 1: Design Features to Manufacturing Features

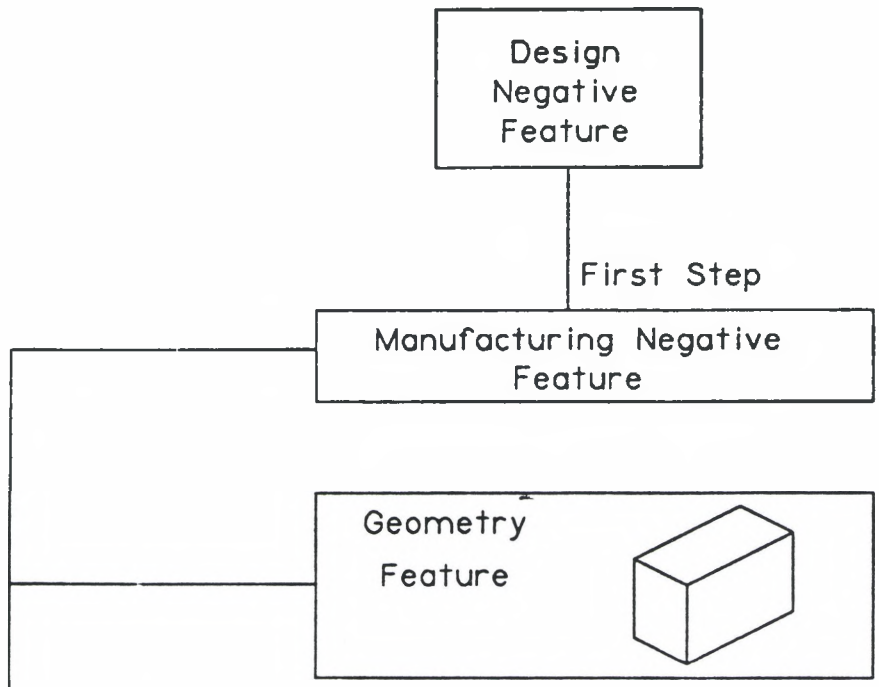
Design	Manufacturing	MetCAPP
Pocket	Pocket	Enclosed Pocket
Through Slot	Through Slot	Through Slot
Edge Cut	Edge Cut	Edge Cut
Open Step	Open Step	Open Step
Corner Step	Corner Step	Corner Step
Blind Hole	Blind Hole	Hole
Through Hole	Through Hole	Hole

In essence, the geometry feature is nothing more than a condensed version of the design feature containing the required information to orient and display the feature in the fabrication environment. The manufacturing feature, in general terms, captures the parameters that affect the

machining process and places them into a form that MetCAPP understands. This form is referred to as the MetCAPP feature. It is important to note that, currently, each manufacturing feature has only one MetCAPP feature associated with it. The NC features associated with a given manufacturing feature are the set of machining operations, returned from the MetCAPP software, that are required to produce that given manufacturing feature. Each step of the operation sequence is an individual NC feature for which NC code will be generated.

For example, a design pocket feature would be translated into a manufacturing pocket feature. This manufacturing feature would be composed of the geometry feature, the MetCAPP pocket feature, and six NC features (See Figure 11). Together, these six features or operations would produce the pocket feature.

To provide the NC code generator with sufficient information, the translation process must be altered in some manner. Referring to Figure 11, the boxes may be viewed as results or objects produced by the translation process which is represented by the connecting lines. From here it should be apparent that the translation process consists of three steps. Therefore, the objective could be achieved by altering any of these three steps of translation. Note that although the NC features are created using the MetCAPP feature, the NC features are attached directly to the manufacturing feature in the same manner as the MetCAPP and geometry features.



Second Step

MetCAPP Feature

DIMENSIONS FOR: Rectangular Pocket - Fully Enclosed

DIMENSION	ENTRY
Length of Pocket	10.000
Width of Pocket	3.000
Depth of Pocket	1.000
Corner Radius	0.250
Fillet Radius	(0.0313)
Maximum Allowable Cutter Diameter	(2.00)
Maximum Height of Obstruction	(0.00)
Thin Wall Condition	(No)
Thin Floor Condition	(No)
Angle Formed Between Floor & Wall	(90.000)
Setup Rigidity (5 - 10, bad to good)	(10)
Sub Floor Length	(0.000)
Sub Floor Width	(0.000)
Sub Floor Axial Depth	(0.000)
Sub Floor Corner Radius	(0.000)
Sub Floor Fillet Radius	(0.000)

Third Step

NC Features

STEP	OPERATION SEQUENCE	TOOLING	PASSES	TIME
1	Plunge End Mill	MLS-0160	1	0.109
2	Slot End Mill	MLS-0160	1	0.817
3	Rough End Mill	MLS-0152	2	0.534
4	Semi-Finish End Mill Wall	MLS-0152	1	0.444
5	Finish End Mill Floor	MLS-0090	3	0.711
6	Finish End Mill Wall	MLS-0211	1	0.619

Figure 11: The Current Translation Process

Altering the third step, however, will not be examined within this paper because the third step of translation is accomplished by the MetCAPP software alone. Thus, there are primarily two methods for accomplishing the objectives of tool selection and NC boundary definition, translational elimination and feature analysis.

Translational Elimination

The first method is to replace the original or design (D1) negative and positive features with a set of negative manufacturing (D2) features such that the summation of negative manufacturing features and the starting block is equal to the summation of design positive and negative features including the starting block.

For example, consider a rectangular pocket with a single rectangular island in it (See Figure 12). The set of design features, pocket-1 and rib-1, is replaced by the set of manufacturing features, pocket-A, pocket-B, pocket-C, and pocket-D. The top part of this figure represents the structure used for associating data. The D2 pocket boxes refer to the manufacturing pocket features that are created in the translation process. The features or sub-features associated with each of these D2 pocket features are represented by the Geom, MCAPP, and NC1 through NC6 boxes.

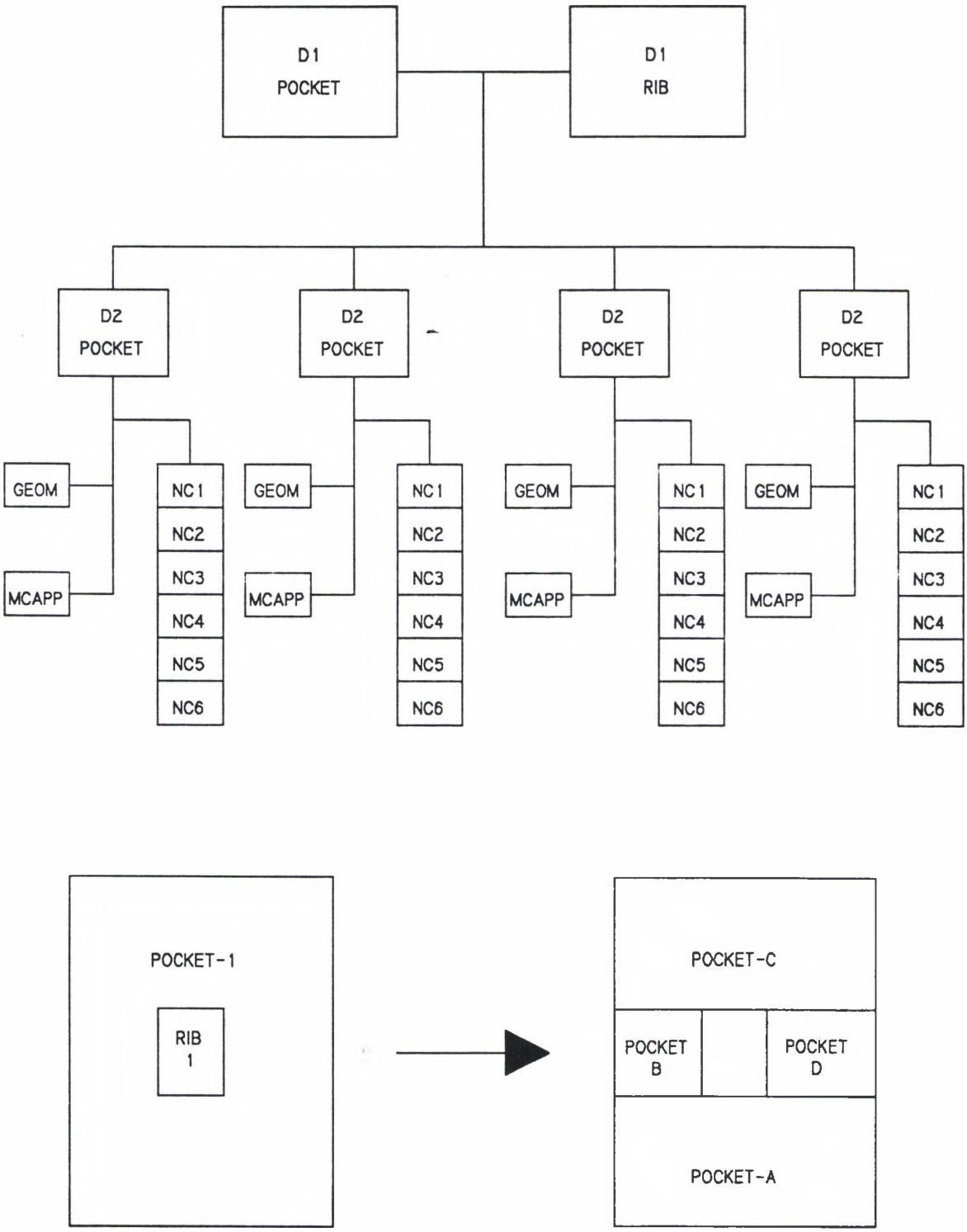


Figure 12: The Method of Translational Elimination

This method is referred to as translational elimination because in the initial step of the translation process, positive features are eliminated. As a result, only negative features exist in the fabrication environment.

By using translational elimination the problem of tool selection is solved since appropriate tooling can be determined independently for each manufacturing pocket feature. Of course, some or all of the related pockets may be able to share common tooling. This would still be addressed by the tooling optimization module that would optimize the tooling across all of the manufacturing features.

In addition to solving the problem of tool selection, translational elimination also solves the problem of NC boundary definition. Since there will not be any positive features in the fabrication sub-system if this method is used for tool selection, there is no longer an issue of NC boundary definition. Each of the newly created manufacturing pockets would inherently have its own boundary.

As might be expected the ability of translational elimination to simultaneously solve both problems is the primary advantage of this method. Unfortunately, there is a major problem with the simplistic method of translational elimination. This is the problem of disassociation of the related manufacturing features. Disassociation would occur because the only means to determine if two given manufacturing features are related would be to examine the design feature or

features that led to their creation.

This lack of association between features that are related may impose limitations on the benefits of feature based CAD systems. Relative to the RDS, disassociation may impair the ability of the Episoidal Associative Memory to form design rules based on manufacturing feedback.

Furthermore, additional work would have to be done to increase the efficiency of the machining plan. Primarily, to eliminate the extra plunge or drill operation on a pocket that is adjacent to another pocket. (For example, pocket-A and pocket-B of Figure 12. These two pocket features could be machined with only a single plunge operation.)

The other problem that would be encountered is the need for a smooth transition between adjacent pockets. If translational elimination is used, the manufacturing pocket features would have to overlap to eliminate the extra corner and fillet material (See Figure 13).

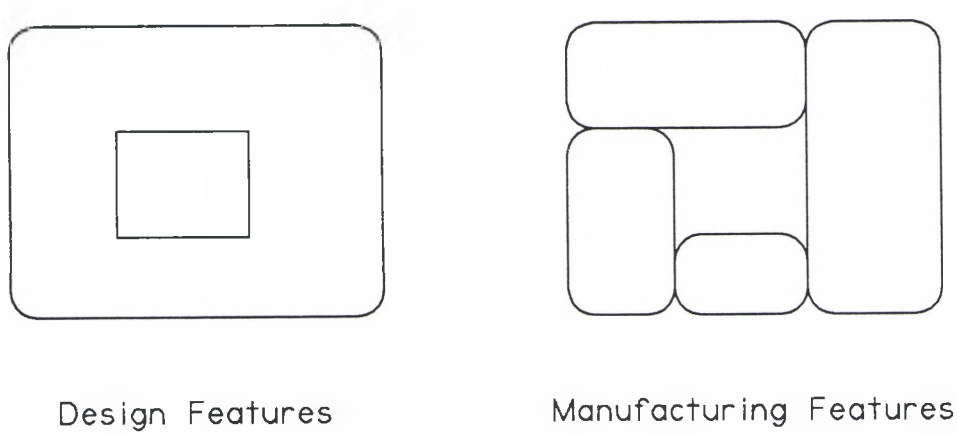


Figure 13: Need for overlapping pockets

Feature Analysis

Returning to Figure 11, if the first step of the translation process is not altered and the third step of the translation process is not alterable, then only the second step remains. In general, altering the second step of the translation process, the creation of the MetCAPP feature, is referred to as the method of feature analysis.

Primarily, there are two methods for accomplishing feature analysis. Both of these methods involve, at least potentially, the creation of multiple MetCAPP features. Since the first step of the translation process is unaltered, each corresponding negative design feature is still translated into a single negative manufacturing feature. This avoids the problem of disassociation which is encountered in translational elimination.

Mathematical Analysis. Feature analysis can be performed mathematically by developing a similarity function, f, which uses the machining parameters as variables. Then, by calculating the function at different locations in the corresponding negative feature, comparisons could be made and on the basis of these comparisons, a variable number of MetCAPP features could be created.

For pocket features on the same part, tool selection depends on five variables: length, width, height, corner radius, and fillet radius. Since the similarity function would be operating on the parameters of a single pocket, it is

possible to narrow the function to include only the length and width dimensions because the depth, corner radius, and fillet radius will be constant for any given pocket in the current RDS.

Using this function, the regions of a pocket-island interaction would be similar if the values of the function were within some hypothetical interval, delta, of each other. For each set of similar regions, one MetCAPP feature would be created to determine the tooling and NC features (operations) required for those similar regions.

Having established which regions are similar and having determined the tooling for each region, NC boundaries can now be defined. To define the NC boundaries, the overlap of regions one through four must be eliminated. If the two intersecting regions are similar, then the intersection may be subtracted from either region. Otherwise, the intersection must be subtracted from the region that will be machined with the smaller tool diameter (See Figure 14).

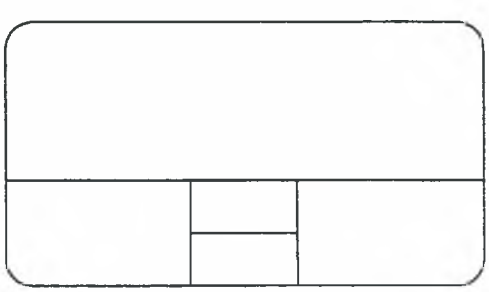


Figure 14: NC boundary definition

The advantage of mathematical feature analysis is that MetCAPP features would be created only as needed. Most pocket-single island interactions could probably be handled with one or two MetCAPP features. This would reduce the number of calls to the MetCAPP software and significantly shorten the processing time of positive-negative feature interactions.

Unfortunately, the entire method depends on developing an accurate and suitable similarity function. Without the ability to examine the internal workings of the MetCAPP software, this would not be an easy task. Furthermore, using mathematical feature analysis more accurately addresses the problem of tooling optimization than that of tool selection for positive-negative feature interactions.

Assuming a genuine need to perform tooling optimization across all of the manufacturing features of a given part, using the same similarity function to process the positive-negative interactions would be somewhat redundant and inflexible. For these reasons, a different variety of feature analysis was chosen as the final solution to be implemented within the RDS. Although still imperfect, it possesses several strengths.

Mapping Analysis. To perform mapping feature analysis, the resulting negative volume of a pocket-island interaction is represented as the union of MetCAPP features. In the case of a single island in a pocket, this involves a very

straightforward, simplistic approach.

For example, the design features in Figure 15 are replaced by a single manufacturing pocket feature that has four MetCAPP features associated with it. Note that where the MetCAPP features intersect, the tooling of either feature may be selected to remove the material in that intersection. This is to insure the efficiency of the process plan.

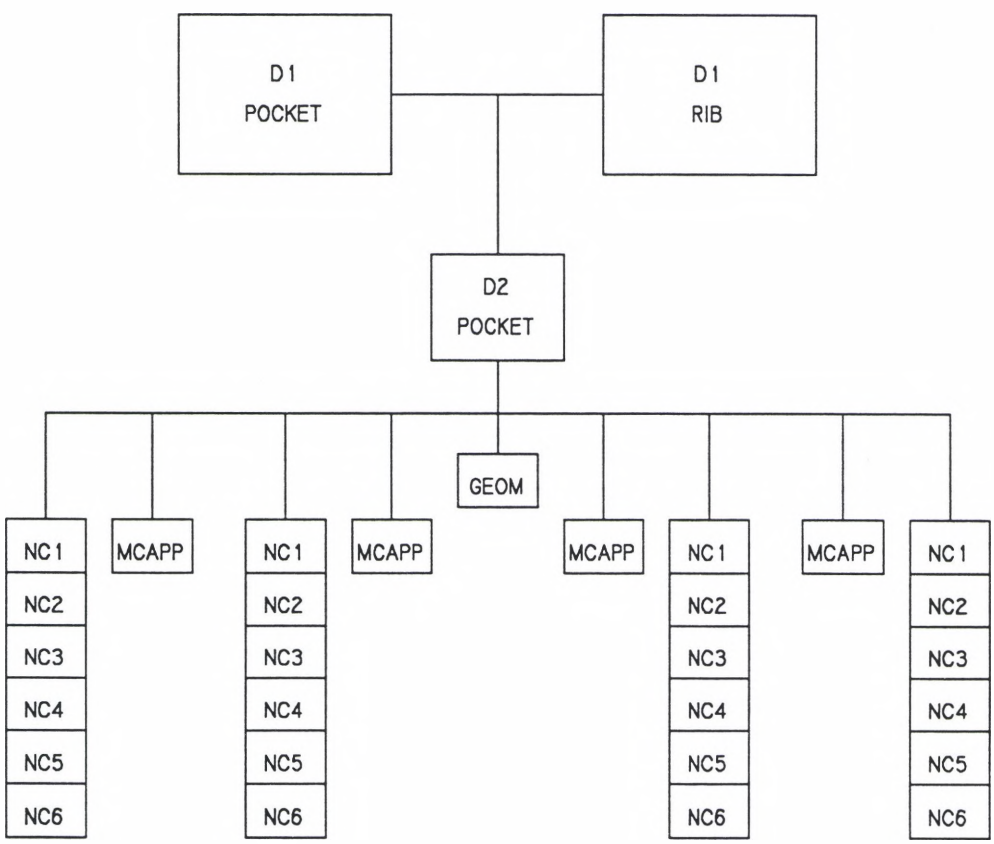


Figure 15: Feature Analysis Translation Process

Then, in the same manner as before, the NC boundaries can be defined (See Figure 16). By specifying which boundaries can be violated by the milling cutter, a smooth transition between machining regions will be insured.

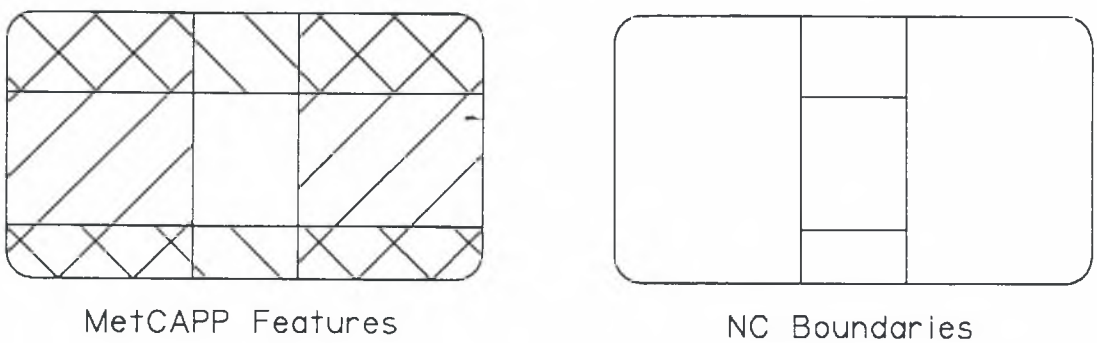


Figure 16: MetCAPP Features to NC Boundaries

One attribute of this method which may appear to be a disadvantage is the need to create a large number of MetCAPP features that will probably be very similar. Currently, this requires many calls to the MetCAPP software and represents a substantial time loss in the processing of positive features. However, when a tooling optimization module is integrated with the fabrication sub-system of the RDS, the MetCAPP features for the entire part could be checked for similarity, thereby reducing the number of calls to MetCAPP. This would allow the tooling optimization to be more accurate because a tool that might not be justifiable for use in a single region of a single pocket may now be justified because it will provide a time savings on other regions of other pockets.

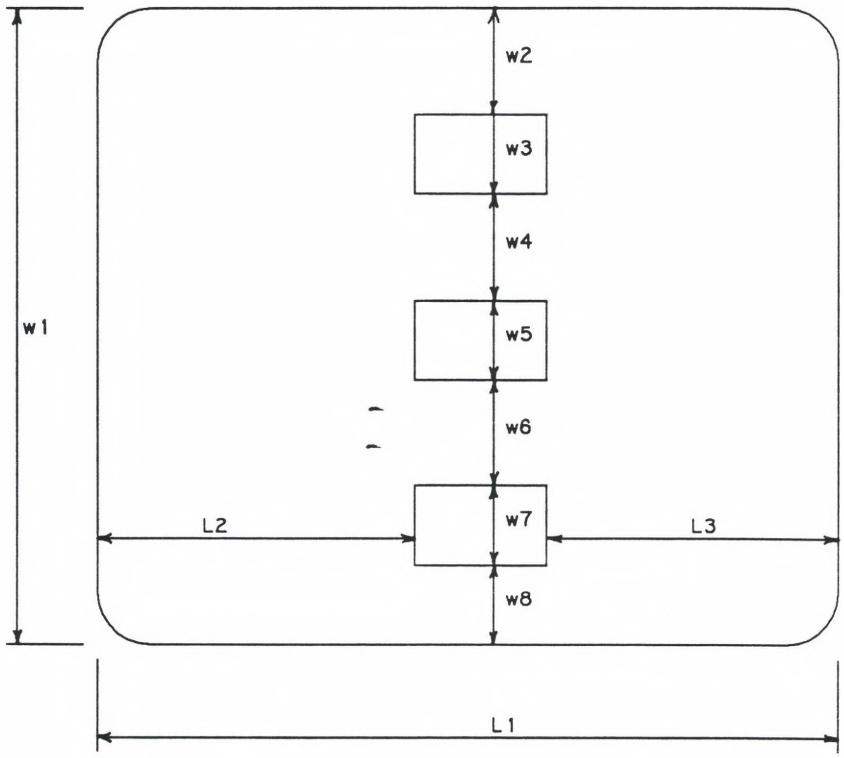
For example, consider a pocket feature with an island that may be machined with two different tools in 20 seconds of cutting time and another 10 seconds for the tool change. The feature could also be machined using only one of the tools in 25 seconds of cutting time. Obviously, if this is the only feature being considered, it is more efficient to use the single tool alternative for a 5 seconds time savings. If, however, there were six identical features of this kind, it would be more efficient to use the two-tool alternative for a time savings of 20 seconds assuming that the tool is already in the magazine (130 seconds for two tools; 150 seconds for one tool).

Multiple Islands

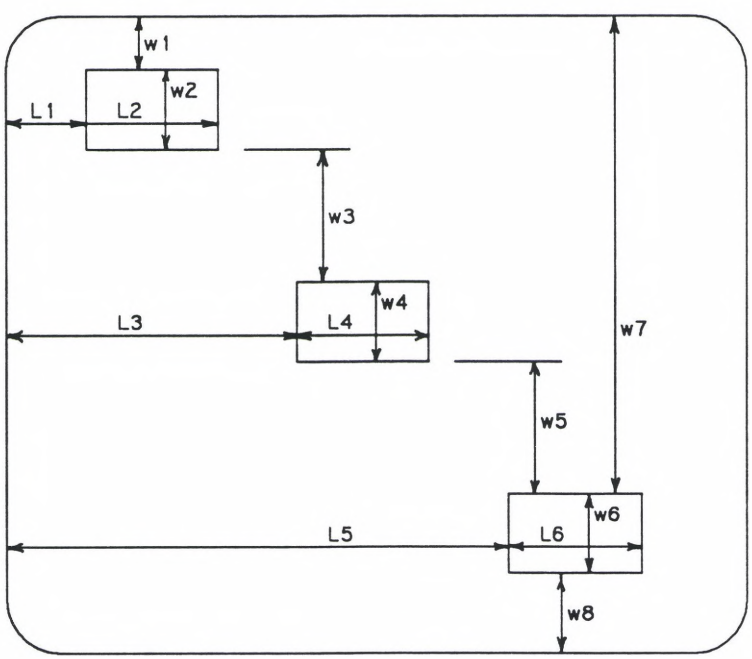
The ability of feature analysis to analyze and process pocket-single island interactions is clearly insufficient to establish it as a solution. To be considered a solution, feature analysis must be capable of processing pocket-multiple island interactions in a logical manner. To say the least, automating feature analysis to process multiple island interactions is not trivial.

Tool Selection

The problem is that analyzing a single island in a pocket is very straightforward, involving only six dimensions, whereas three or even two islands in a pocket not only



(a)



(b)

Figure 17: Critical Dimensions for Island Arrangements

introduce more dimensions, they introduce an undetermined number of non-zero dimensions.

This is illustrated with the help of Figure 17 which depicts two possible arrangements of three islands in a pocket. With a little time, one should conclude that, indeed, all of the dimensions of Figure 17b are needed. All of these dimensions are required to achieve the goal of feature analysis--to determine the room available, in terms of length and width, for a cutting tool at every location in a pocket-island interaction.

Returning to the case of a single island in a pocket, consider the impact the intersections of the MetCAPP features have on the problem. The intersections lead to the concept of imposing a three by three non-uniform grid onto the geometry of the interaction (See Figure 18).

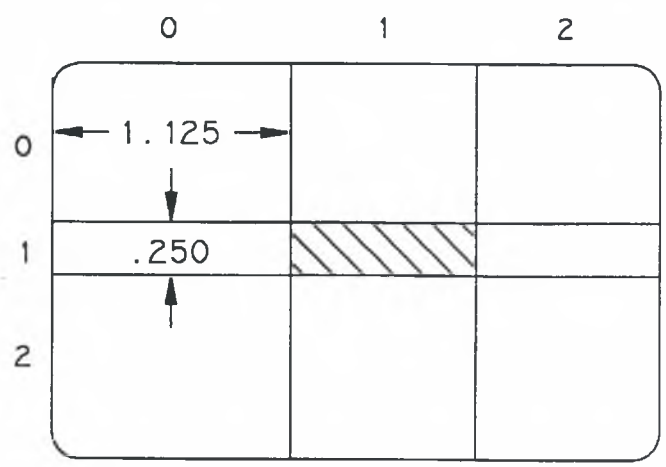


Figure 18: Imposing a Non-Uniform Grid

The center of the grid, rectangle (1, 1), represents the positive volume of the island and will not be removed in the machining process. The remaining eight rectangles, representing the negative volume of the interaction, will be removed in the machining process.

To determine the maximum amount of room available for a cutting tool, the rectangles of the grid are combined to form larger rectangles. This is important because the length and width of each individual rectangle may not accurately define the room available for a given tool.

This is certainly the case for rectangle (1, 0). Initially, the width of this rectangle restricts the size of a tool to 0.25 inches. Clearly, however, a 1.0 inch tool would be acceptable to remove the material between the left wall of the pocket and the left side of the island.

To consider the more complex case of two islands in a pocket, again impose a grid on the resulting geometry. After imposing a grid, create MetCAPP features by combining grid rectangles to form larger rectangles. The process used to develop these larger rectangles is not critical provided that every basic grid rectangle is associated with at least one MetCAPP feature and the rectangles are expanded to their maximum rectangular size.

An example of the results produced by applying this procedure for the interaction depicted in Figure 19, is shown in Table 2. With the entire negative volume of the interaction

mapped into MetCAPP features, the task of tool selection is accomplished.

Table 2: Composition of MetCAPP Features

MetCAPP Feature	Composition
1	(0 0) (0 1) (0 2) (0 3) (0 4)
2	(0 0) (1 0) (2 0) (3 0) (4 0)
3	(2 0) (2 1) (2 2) (3 0) (3 1) (3 2) (4 0) (4 1) (4 2)
4	(4 0) (4 1) (4 2) (4 3) (4 4)
5	(0 2) (0 3) (0 4) (1 2) (1 3) (1 4) (2 2) (2 3) (2 4)
6	(0 4) (1 4) (2 4) (3 4) (4 4)

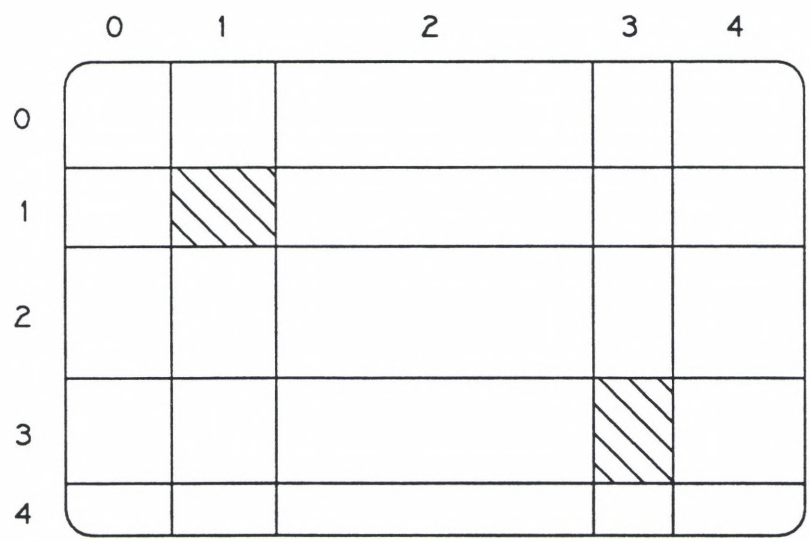


Figure 19: Multiple islands in a pocket

NC Boundary Definition

The NC boundaries are established for multiple island interactions similarly to the way in which they are defined for single island interactions. When a rectangle is associated with more than one MetCAPP feature, it is removed from the MetCAPP feature that produced the tool with a smaller diameter. This insures that every rectangle of the grid is actually machined with the largest available tool. If, upon removing a rectangle from a MetCAPP feature, that MetCAPP feature becomes non-rectangular, then the feature will be reduced to form new NC boundaries that are rectangular.

The rectangles associated with each bounded machining region of the interaction depicted in Figure 19 are listed in Table 3. Note that machining regions 5a, 5b, and 5c are the results of reducing MetCAPP feature No. 5.

Table 3: Composition of NC Machining Regions

Machining Region	Composition
1	(0 0) (0 1)
2	(1 0)
3	(2 0) (2 1) (2 2) (3 0) (3 1) (3 2) (4 0) (4 1) (4 2)
4	(4 3)
5a	(0 2) (1 2)
5b	(0 3) (0 4) (1 3) (1 4)
5c	(2 3) (2 4)
6	(3 4)

CHAPTER III

FEATURE ANALYSIS

After reviewing the available methods for processing positive-negative feature interactions and selecting feature analysis by mapping, the computer code to automatically process islands in pockets was developed and integrated with the RDS. In this chapter, the general structure and workings of this code will be addressed. Not only should this provide some general knowledge of the code, it should facilitate a greater understanding of the mechanics involved in the application of feature analysis.

It should be noted, however, that the code which has been developed involves two major assumptions. First, it is assumed that the fillet radius of the pocket feature and the fillet radii of all interacting rib features are equal. Second, the corner radii of all interacting rib features are assumed to be equal to zero.

These assumptions do not overly limit the practical application of this code. Further, until NC code is generated from the information produced by this code, it should remain simple to reduce the amount of any fine tuning that may be required.

Overview

The entire RDS, with the exception of the MetCAPP software package and interface, is written in Lisp on top of a Concept Modeller Lisp template. To further explain this, the Concept Modeller, a Wisdom Systems product, is an object-oriented CAD system designed to be customized by its user. The RDS is one example, although a highly unique one, of this customization. The Concept Modeller Lisp template refers to all of those Lisp functions that are not ordinarily available in Common Lisp.

In the RDS, a feature such as a pocket is an object with associated properties such as depth, width, height, etc. To process islands in pockets, seven additional properties are added to the D2-pocket-feature. These seven new properties exist for all pockets in the fabrication sub-system regardless of whether or not they are impacted by positive features. By structuring the code in this manner, a consistency is established to avoid the further complication of those tasks performed by other modules of the fabrication sub-system.

The end goal of all of this work is to communicate to the NC code generation module the material to be removed to produce a given pocket feature. With or without positive features, there is by definition material to be removed for every pocket feature in the fabrication sub-system.

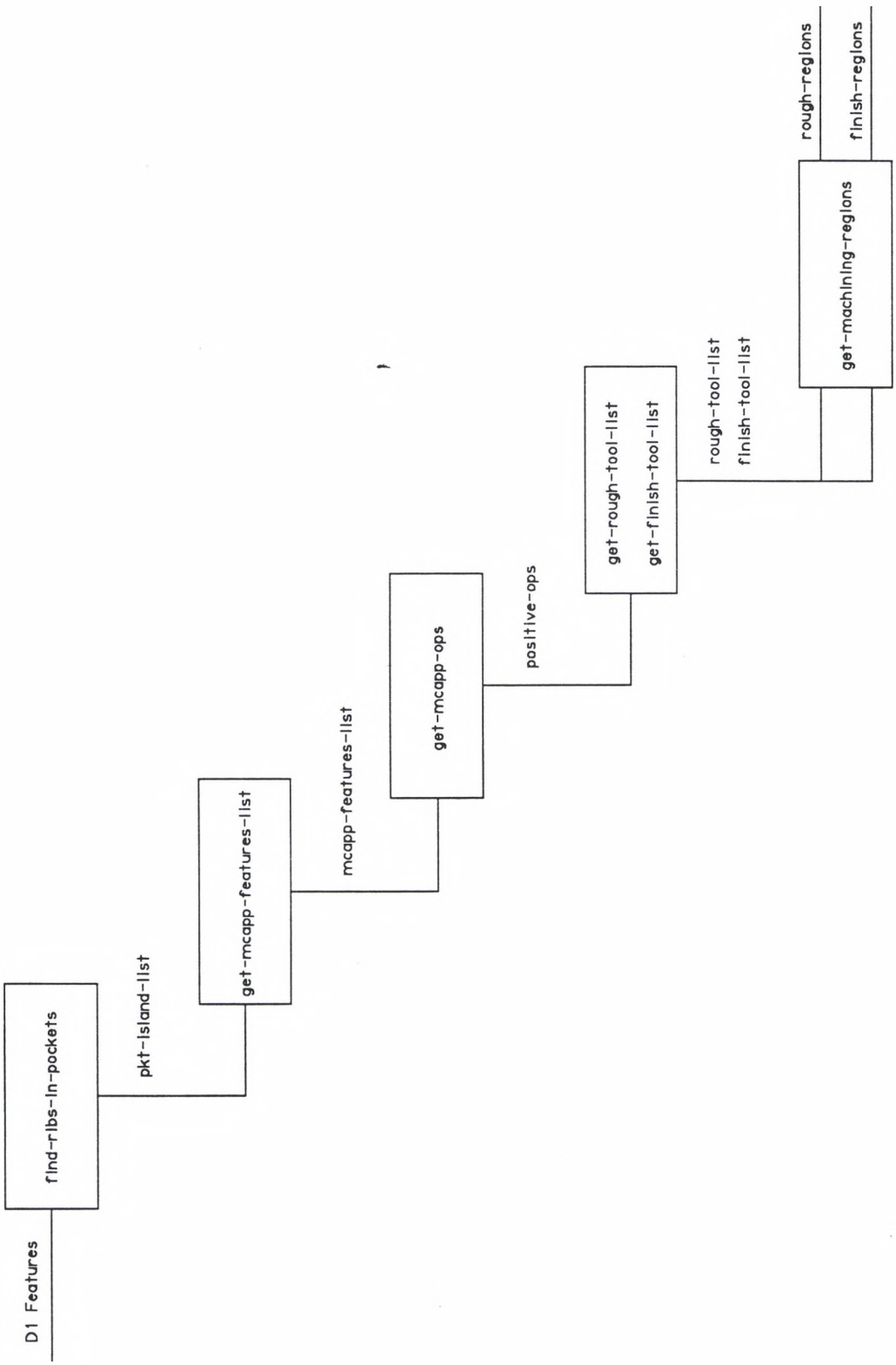


Figure 20: An Overview of the Positive Features Code

The names of these seven new properties as well as an overview of the code are shown in Figure 20. Note the dependence of the properties on those that have already been defined. A definition and explanation of the properties will be given in the following sections. The code to define each property is contained in Appendix A through Appendix F. Each appendix contains all of the functions required to define a property.

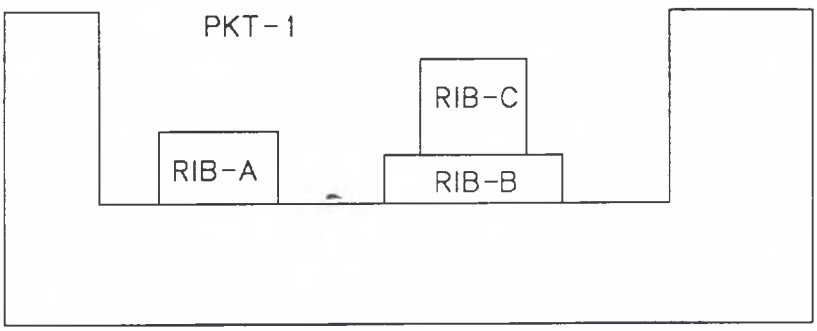
Positive Feature Recognition

The key to the entire process of dealing with positive-negative feature interactions is to first identify them in a meaningful fashion. The result of this detection process is the Pkt-Island-List property of a pocket. This list identifies the pocket, the ribs impacting it, and the height of the layer for each layer of the interaction (See Figure 21).

To compile this list, an extremely thin, three-dimensional plate geometry is created at the bottom of the pocket. The AGM solid modeller is then used to check each rib feature of the part to determine if it intersects the plate. This establishes the bottom layer of the pocket which will become the last element of the Pkt-Island-List.

The preceding elements of the list are then determined by moving the thin plate to the top of the shortest rib of that layer. Referring to Figure 21, the plate is moved to the top

of rib-B, then to the top of rib-A, and finally to the top of rib-C.




```
( (PKT-1      (NIL)       0.250)
  (PKT-1      (RIB-C)     0.375)
  (PKT-1 (RIB-A RIB-C) 0.125)
  (PKT-1 (RIB-A RIB-B) 0.250) )
```

Figure 21: The Pkt-Island-List

Building MetCAPP Features

After establishing the Pkt-Island-List, the pocket-island interaction can be processed one layer at a time. The first step in processing these layers is to develop the MetCAPP-Features-List where each element of the list contains the MetCAPP features of a single layer of the interaction.

To begin building the MetCAPP feature that will compose this list, a non-uniform grid is imposed on the interaction such that each rectangle of the grid is either completely negative or completely positive volume. The negative volume is then mapped into MetCAPP features with each rectangle being

mapped into at least one feature and each feature retaining a rectangular geometry. This is accomplished by starting with any rectangle that has not yet been mapped and expanding it in the smaller of its two dimensions, length and width. The expansion process stops when the MetCAPP feature is in contact with positive volume on all four sides.

Referring to Table 4 for a list of properties of a MetCAPP feature, the length and width are determined by the previously described expansion process. The height is obtained from the height of the layer, contained in the Pkt-Island-List. The maximum height of obstruction is equal to the summation of the height of each layer above the given one plus the maximum height of obstruction associated with the D2-pocket-feature. Then, with the exception of the sub-floor properties which are

Table 4: MetCAPP Properties for a Pocket

Length of Pocket
Width of Pocket
Depth of Pocket
Corner Radius
Fillet Radius
Maximum Allowable Cutter Diameter
Maximum Height Obstruction
Thin Wall Condition
Thin Floor Condition
Angle Formed Between the Floor & Wall
Setup Rigidity (5-10, bad to good)
Sub Floor Length
Sub Floor Width
Sub Floor Axial Depth
Sub Floor Corner Radius
Sub Floor Fillet Radius

always set equal to zero, the remaining properties of the MetCAPP feature are set equal to the same properties associated directly with the D2-pocket-feature being processed.

Note that the maximum height of obstruction property prevents MetCAPP from returning a tool to short to mill the pocket. If this property is not properly set, the machine collet could collide with the part. MetCAPP will also compensate for a long tool by reducing the feed rate to prevent excessive tool deflection. In the event that a suitable tool cannot be found because of the height to width ratio of the negative volume, MetCAPP returns a warning message.

In the future when the restrictions requiring the pocket fillet radius to be equal to the rib fillet radii are removed, the largest fillet radius of any rib in the pocket or the fillet radius of the pocket itself will be used for all of the MetCAPP features of that layer. Then, additional MetCAPP features are created as needed to determine the required finishing tools establish the proper fillet radii.

Getting the Operations

To get the machining operations associated with the MetCAPP features, each feature is sent one at a time through the interface to MetCAPP. The operations are then received back through the interface. Each set of operations then

becomes a single sub-element of a list. All of the sub-elements associated with a particular layer form an element of the positive-ops property. Most of the functions to accomplish this task are already present in the fabrication sub-system.

The most important part of this process is that the operations lists received from MetCAPP most be assembled such that the elements of the positive-ops list of each pocket exactly match with their corresponding elements of the MetCAPP-features-list for that pocket. If their would be a failure to achieve this, the rest of the processing would break down.

Establishing Tooling

The set of operations returned from the MetCAPP software generally consists of six operations or steps to create that given feature. Currently, there is considerable debate and uncertainty about the need to completely follow these steps verbatim. In particular, many of these steps require a separate tool which typically results in a total of four different end mills to perform all six steps. As a result, the positive feature code selects only two tools from each set of operations in an attempt to simplify the positive features code and streamline the operations. In the future, after a more thorough investigation of the MetCAPP process plan, the

positive features code can be modified to include more or all of the tools or operations with a minimal programming effort.

The two tools that are selected are the plunge end mill and the wall finish end mill. These tools correspond to the first and last tools of the operations list, respectively, and will be used for roughing and finishing, respectively. To facilitate the necessary data handling, two properties are assigned to the D2-pocket-feature, the rough-tool-list and the finish-tool-list, to store the tool identification code associated with each tool.

This is probably not the optimum solution to the problem. Unfortunately, until the fabrication sub-system is complete and parts are actually machined with NC code generated by the RDS, fine tuning of the positive features code is virtually impossible. With the flexible overall structure of the highly modular code, however, it will require relatively minimal programming effort to select more or all of the tools recommended by MetCAPP.

The Machining Regions

The final step in processing pocket-island interactions is to transform the MetCAPP features into machining regions and provide the NC generation module with the tooling, entry and exit, and side violation information. All of this information is produced by a single set of Lisp functions that define the rough-regions and the finish-regions properties

with the rough-tool-list and the finish-tool-list as part of the input.

To assist in the explanation of the activities performed to produce either the rough-regions or the finish-regions, please refer to Figure 22 which depicts a single element of the rough-regions property of a given D2-pocket-feature. Each element of the list that composes this property contains all of the necessary information for a single layer. The top layer of a pocket-island interaction is represented by the first element of the list and the bottom layer is represented by the last element.

```
( "MLS-0169"
 ((17.0 9.0) (19.0 9.0) (19.0 12.0) (17.0 12.0)
 (BACK LEFT FRONT)
 ((18.0 8.449) (18.0 9.551) (18.0 10.5)) )
```

Figure 22: An Element of the Rough-Regions Property

The first critical step in compiling the information to be associated with an element of the rough-regions list is to transform each of the MetCAPP features of a layer into a machining region. To do this, the intersection of the two features must be subtracted from one of the intersecting features. If the tool diameters associated with the two intersecting features are equal, the choice is considered to be arbitrary. Otherwise, the intersection is subtracted from the MetCAPP feature with the smaller tool diameter. Through

this procedure amount of material removed by the larger diameter tools will be effectively maximized.

In some instances, the subtraction of the intersection from a MetCAPP feature will leave a non-rectangular geometry. When this happens, the resulting region will be decomposed into new rectangular regions. By maintaining rectangular machining regions, additional complexity relative to the process of generating NC code is avoided.

After determining the machining region, the four vertices of the region are transformed from the Concept Modeller coordinates to APT coordinates in a standard coordinate system transformation (See Figure 23). The APT coordinate vertices are then placed in a list starting with the front left and proceeding in a counter-clockwise direction. This vertice list in conjunction with the depth of cut which is obtained from the Pkt-Island-List, now defines the volume of material to be removed.

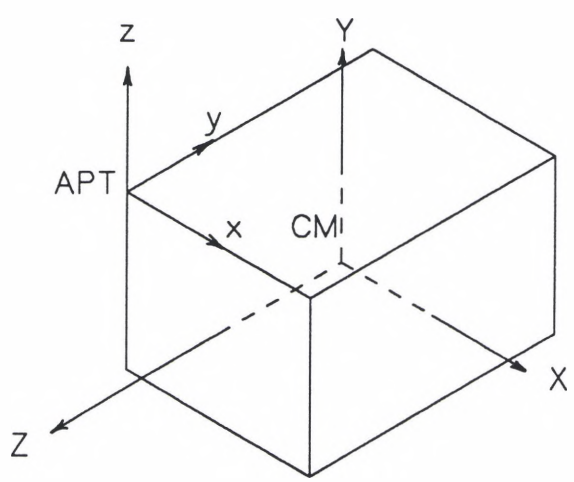
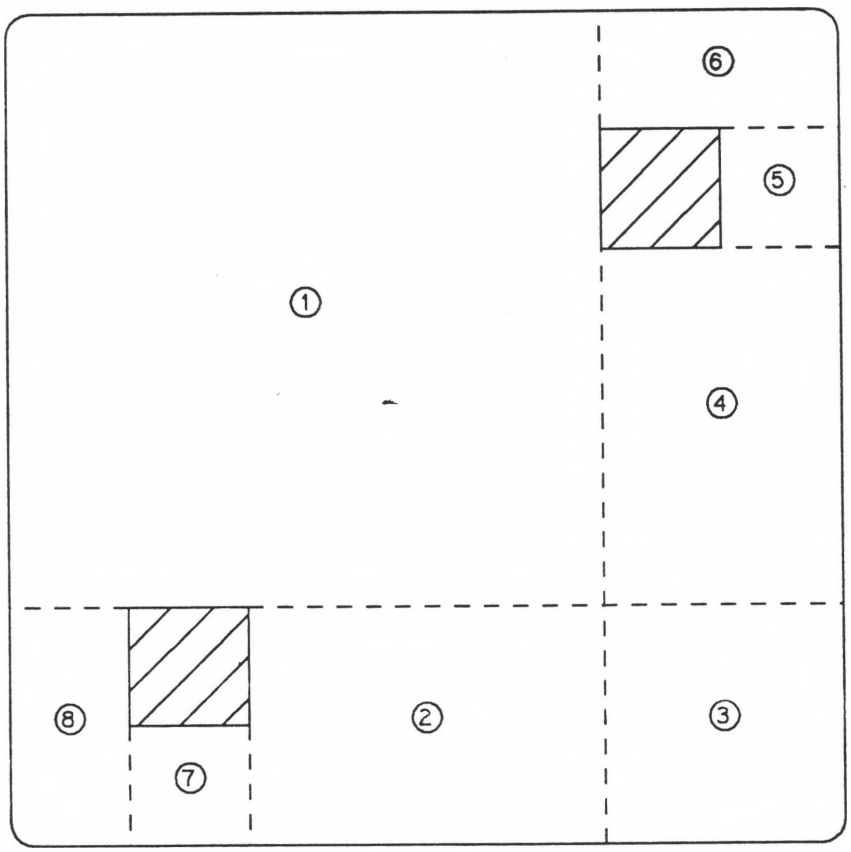


Figure 23: The CM to APT Coordinate Transformation

Also following the determination of the machining region, the machining regions which are adjacent to each region are found. By defining which side the adjacent region is in contact with the region in question, it is possible to define which sides of each machining region may be violated to move the tool from one region to the next and which sides must be violated in order to establish a smooth transition across regions.

Once this information is organized, the machining regions can be efficiently sequenced. This is accomplished with the use of a fairly simple algorithm. The algorithm begins by grouping the regions according to the tool that will be used for that region. Then, beginning with the group of regions to be machined by the largest diameter tool, a region is selected to be machined. Next, any region adjacent to it that is within the same group will be selected. This process continues until none of the remaining regions in the group is adjacent to any of the previously machined regions. This process is then repeated for each group of regions in descending order of tool diameter. Note that in each successive group of regions, the next region to be machined may be adjacent to a region of another group that was already processed. This entire process is illustrated in Figure 24.

Note that the next list is a list of adjacent regions to be considered for the next region to be machined. This list includes all of the regions that are adjacent to the current



Tool Groups: MLS-0169: 1, 2, 3, 4
MLS-0165: 5, 6, 7, 8

CURRENT	NEXT LIST	GROUP LIST
Pocket-1	2, 4, 6, 8	2, 3, 4
Pocket-2	3, 4, 7, 6, 8	3, 4
Pocket-3	4, 7, 6, 8	4
Pocket-4	5, 7, 6, 8	5, 6, 7, 8
Pocket-5	6, 7, 8	6, 7, 8
Pocket-6	7, 8	7, 8
Pocket-7	8	8
Pocket-8	nil	nil

Figure 24: The Sequencing of Machining Regions

region being machined regardless of which tool group they are in. If, however, the first region in the next list is not an element of the current tool group list, it will be skipped

until all of the regions of the current tool group list are machined.

At the same time the regions are sequenced, the tool entry sequence into the region to be machined is determined. The procedure for generating the tool entry sequence, as shown in Figure 25, is to first find the midpoint of the line segment that is common to both machining regions. By adding the product of the inward normal unit vector and the scalar tool radius to this point, the exit point of the just machined region and the entry point of the next region to be machined are established. The third point in the sequence is the center of the next region to be machined. With all of this information, the task of NC code generation for the island-pocket interaction should now be possible without any unnecessary complications.

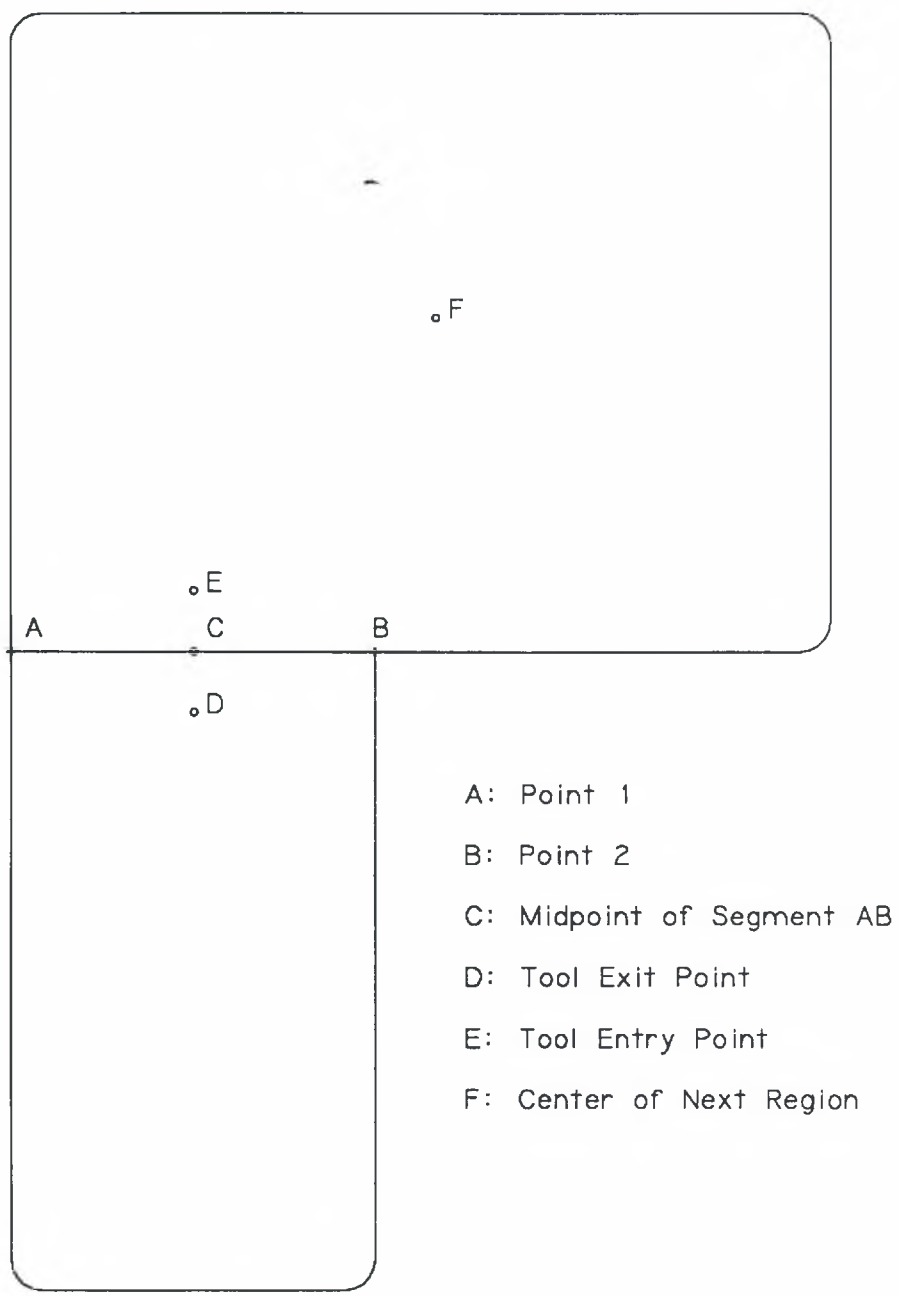


Figure 25: The Tool Entry/Exit Sequence

CHAPTER IV

ADDITIONAL ISSUES

In addition to developing the algorithms and Lisp code that have been described in the preceding sections, the research involved a certain amount of speculation relative to the solutions for items not completely addressed by the research. The results of this speculation relative to several closely related issues are addressed in the following sections.

Miscellaneous Pocket-Island Issues

As has been previously mentioned, the Lisp code developed to handle rib features interacting with pocket features requires two assumptions that limit the practical application of the code. The two assumptions are that the fillet radii of all interacting ribs are equal to the fillet radius of the pocket and that the corner radii of all of the rib features are equal to zero. Although these assumptions place significant restrictions on the use of the positive features code, the modifications required to remove these restrictions are not major.

Non-Zero Rib Corner Radii

Perhaps the greatest restriction placed on the positive features processing code is the requirement that the fillet radius of all interacting rib features be equal to zero. To eliminate this requirement, it is necessary to communicate to the NC code generation module the corner radius and the center of curvature as well as the appropriate tooling.

Relative to the tooling, a decision has to be made as to whether or not a roughing pass will be needed in addition to the finishing pass. Probably the most convenient way to make this determination would be to compare the corner radius to tool radius ratio to some critical value:

(Corner Radius) / (Tool Radius) ? K

If K, is exceeded, indicating a relatively large amount of material to be removed by the finishing tool, a roughing pass is required. The majority of the work required to eliminate the corner radius restriction involves the NC code generation.

Unequal Fillet Radii

The elimination of the requirement that the fillet radii of the interacting rib and pocket features be equal involves some what more work, but is still reasonably simple. The desired fillet radius of a machining region will impact the finishing tool and possibly the roughing tool selection.

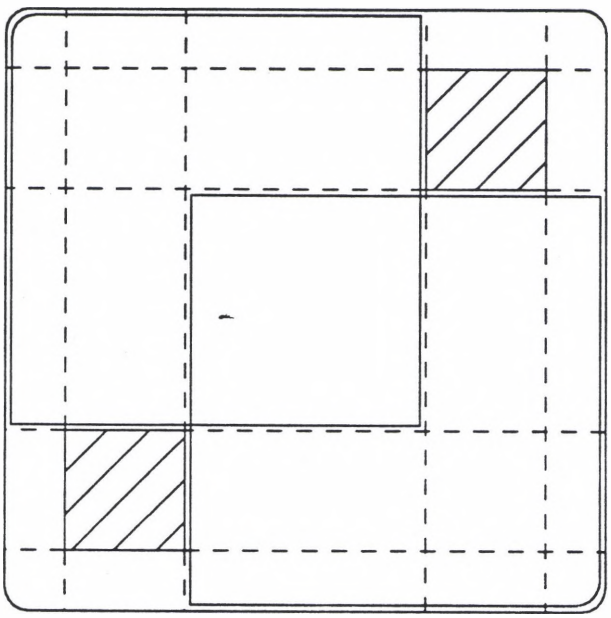
To account for this, the largest fillet radius of the layer being processed should be used with all of the MetCAPP

features of that layer. This will provide the appropriate roughing tools for the layer and also the appropriate finishing tool for the pocket or rib feature associated with that fillet radius. Then, to select the necessary finishing tool or tools for the remaining features of that layer, an additional MetCAPP feature is created with the required fillet radius for each different fillet radius. Finally, the machining region for each extra finishing tool can be defined simply as a rectangle that must be "traced" with tool to produce the proper fillet radius.

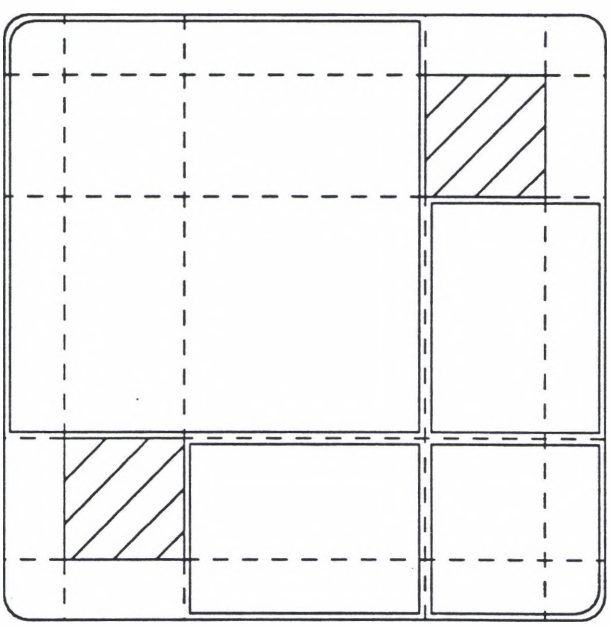
Multi-Sided Convex Machining Regions

Although not necessary, the inclusion of multi-sided, convex machining regions can in some cases result in significant time savings. In particular, these regions are very helpful in the reforming of intersecting regions (See Figure 26).

Currently, when two MetCAPP features intersect, the entire intersection is subtracted from one of the two corresponding machining regions. In some instances, this results in one of the corresponding machining regions becoming non-rectangular which requires it to be further decomposed into a set of rectangular machining regions. The operations to machine the set of new machining regions will sometimes be more time consuming then to machine the entire region as it existed prior to the subtraction of the intersection.



(a) Intersecting MetCAPP Features



(b) Resulting Machining Regions

Figure 26: The Processing of Intersecting MetCAPP Features

To overcome this inefficiency, the rectangles composing the intersection could be split between the two intersecting MetCAPP features to form two five-sided-regions (See Figure 27). Using this procedure, time savings could be increased and the multi-sided, convex machining regions should not pose any major problem to the automated process of generating NC code.

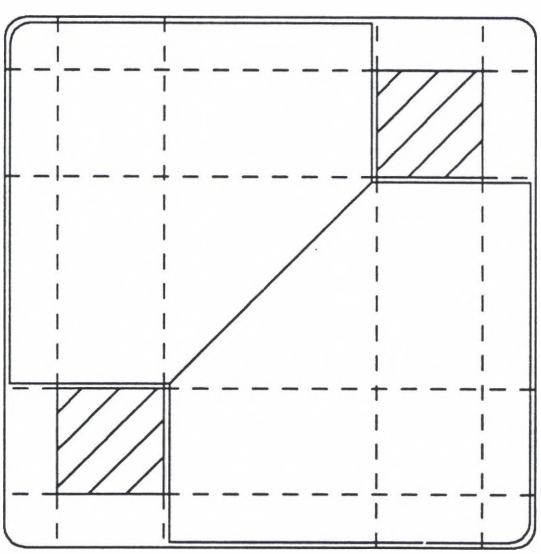


Figure 27: Multi-Sided Machining Regions

Non-Aligned Ribs

Although the RDS does not currently enable a user to rotate a rib to a position in which it is no longer aligned within a pocket feature, this capability will certainly be forthcoming. Another reason for addressing this topic is that the related issues of machining triangular and quadrilateral pockets can also be examined.

In examining a non-aligned rib feature interacting with a rectangular pocket feature, it should be obvious that the corner location alone will be of little value since it may be outside of the rectangular pocket feature (See Figure 28). Instead it will be necessary to establish a modified corner location based on the original location and the radius of curvature of the corner.

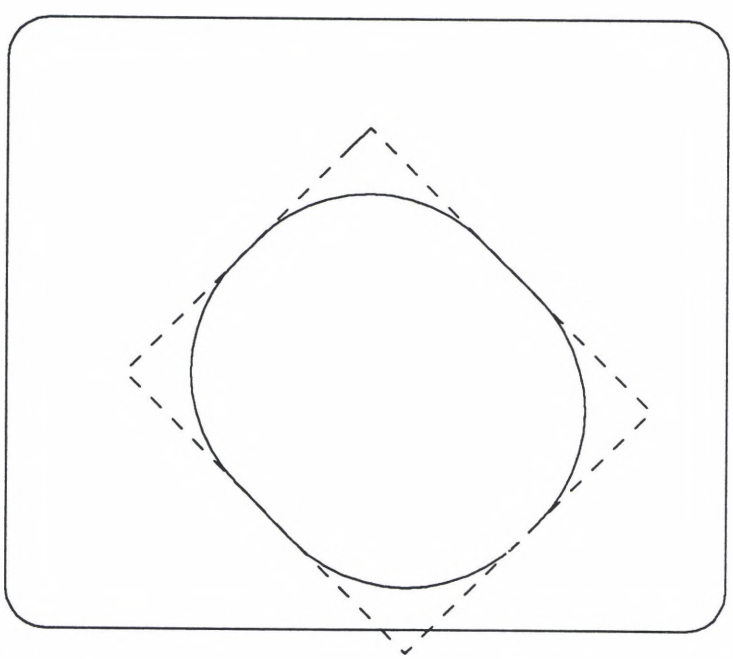


Figure 28: Non-Aligned Rib Feature in Pocket

The geometry resulting from this interaction would appear to easily lend itself to the use of triangular pocket MetCAPP features for mapping. Unfortunately, the MetCAPP software package does not currently provide such a feature.

Multi-Sided Pocket Features

The last of the related issues to be addressed within this paper is the issue of multi-sided design pockets in either a convex or a concave configuration. Again this issue involves a capability that is not yet available in the feature based design environment of the RDS. As a result the specifics of how to provide the necessary information to the NC code generation module cannot be addressed as they will depend heavily upon the feature representation of such a pocket.

Depending on whether or not the pocket is concave or convex it could be handled in one of two ways. If the pocket is convex, mathematical analysis may be used to determine the maximum rectangular region to be used as the MetCAPP feature within the pocket (See Figure 30). Then, the convex pocket feature could be used for NC boundary definition. There is no need to alter the feature because, as was previously mentioned, there are no major obstacles to be overcome with regard to automatically generating NC code for a multi-sided convex machining region.

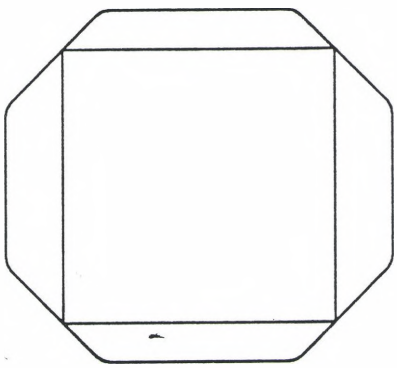


Figure 30: Multi-Sided Convex Pocket Feature

If, however, the multi-sided region is concave, a non-uniform grid could be imposed on the feature and the same mapping procedure already described could be used for tool selection and NC boundary definition. This process could be further complicated if the feature combined multi-sided concave and convex regions, but it is still probably the best alternative.

CHAPTER V

CONCLUSIONS

As the RDS developed, a need was recognized within the fabrication sub-system to develop and implement a procedure to accomplish the tasks of tool selection and NC boundary for the resulting negative volume of a positive-negative feature interaction. After an initial investigation of the general problem, the field of research was reduced to include only pocket-island interactions where the rib features were restricted to an aligned position relative to the rectangular pocket.

The investigation shows that the broadly viewed design to manufacturing translation process is the arena in which to develop the necessary procedure. Being a three-step process, the translation process provides three levels at which modifications can be made to accomplish the desired objectives.

Since altering the third step of translation would require the MetCAPP software package itself to be altered, which is considered to be an alternative beyond the scope of this investigation, this paper explores only two methods. The first alternative, translational elimination, is a process in which the design pocket feature and rib features would be

replaced by a set of manufacturing pocket features thereby eliminating the rib feature in the first step of translation. This process, however, would disrupt the one-to-one correspondence of design features to manufacturing features presenting possible problems in disassociation.

Largely for this reason, the second alternative is chosen as the best process to accomplish the sought after objectives. By decomposing the pocket-island interaction into layers of rectangles, the resulting negative volume can be mapped into MetCAPP features and then into machining regions. The first of these mappings succeeds in accomplishing the task of tool selection and the second adequately establishes the required NC boundaries. Combined, this critical information enables the automatic generation of NC code.

To date, the Lisp code to perform feature analysis by mapping has been developed and is currently being evaluated for single and multiple island interaction.

CHAPTER VI

RECOMMENDATIONS

Despite the tremendous advance in the growing ability of the fabrication sub-system to effectively process positive-negative feature interactions, a great deal of work remains. Foremost among the remaining tasks is the need to adapt the NC code generation module to produce NC code from the information provided in the rough-regions and finish-regions properties of the D2-pocket-feature. To complete this task and fulfill the remaining validation requirements of the positive features code, the NC code should be executed (at least through the use of verification software) and evaluated for a variety of pocket-island interactions.

In the process of this validation, as well as other evaluations or validations of results involving the use of MetCAPP to select tools for multiple features, a tremendous need for tooling optimization across MetCAPP features should become evident. The inability of MetCAPP to select tools based on a set of features instead of on a single feature basis is one of the obstacles that must be overcome to successfully integrate MetCAPP with the RDS.

Regardless of whether this problem is to be solved within MetCAPP itself or merely within the RDS, one of the most

critical factors influencing tool selection should be the number of parts to be produced. As the tool setup time is distributed over an increasing number of parts, the more significant the time savings, in terms of decreased machining time, associated with a given tool will become.

In addition to developing the ability to generate NC code from new properties added to the D2-pocket-feature, there is a need to update the operations sequencing module and the user interface. Since the final output of the positive features code is a description of what material must be removed and how it should be removed, the rough-regions and the finish-regions properties of a d2-pocket-feature will always be defined regardless of whether or not any islands interact with the given pocket. This being the case, the operation sequencing module should be updated so that it always processes the newly defined pocket properties. By doing this, a high level of consistency will be maintained which will benefit the RDS in the long term.

Also the user interface should be expanded to work in conjunction with the positive features code. Cartaya [1] stated the need for an interactive user interface that would allow the user to modify the design to manufacturing feature translation and the tooling and operations produced by MetCAPP. If anything, this need has been magnified by the revised translation process. Clearly, the RDS should extend to the user the ability to modify the actual mapping process

used to determine tooling and boundaries.

In addition to the just mentioned integration work, there is also a need to improve and expand the positive features code itself. Some of the future work was described to varying degrees in the chapter about additional issues. Probably the two most important short term refinements are the need to eliminate the restrictions on the rib corner radii and fillet radii as well as the need to incorporate the ability to define multi-sided convex machining regions when advantageous.

As the RDS continues to expand, additional work will be required in all areas of the fabrication sub-system. The positive features code is no exception to this rule.

BIBLIOGRAPHY

1. Cartaya, Christine Marie. Computer Aided Process Planning (CAPP): The User Interface of the RDS, Masters Thesis, University of Dayton, December 1991.

2. Concept Modeller Reference Manual, Release 1.3, Wisdom Systems; Cleveland, Ohio: July 1991.

3. Dvorak, Paul., "Keeping Talent with Knowledge Systems.", Machine Design: August 22, 1991.

4. Hayes, Caroline., Planning in the Machining Domain: Using Goal Interactions to Guide Search, Masters Thesis, Carnegie Mellon University, April 1987.

5. Karinthi, Raghu, Dana Nau, and Qiang Yang., Handling Feature Interactions in Process Planning

6. LeClair, S. R., "The Rapid Design System: Memory-Driven Feature-Based Design.", Proceedings of the 1991 IEEE Conference on Systems Engineering; Dayton Ohio: August 3, 1991.

7. MetCAPP User's Guide Release 2., Institute for Advanced Manufacturing Sciences, Cincinnati, Ohio: September, 1990.

APPENDIX A

PKT-ISLAND-LIST COMPUTER CODE

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;                                                   ;;;;;;;;
;;;;;;;;       FUNCTIONS TO DEFINE PKT-ISLAND-LIST         ;;;;;;;;
;;;;;;;;                                                   ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; This is the parent function controling the other functions
;;; that are called to create the pkt-island-list for a
;;; d2-pocket-feature. It works by establishing the first
;;; intervals for a thin plate geom created by the agm.
;;;
;;; input:  pkt   The design pocket feature corresponding to the
;;;               d2-pocket-feature being translated.
;;;
;;; output: The pkt-island-list is returned.
;;;         ((pkt (rib-list) height) (pkt (rib-list) height) ...)
;;;
(defun find-ribs-in-pocket (pkt)
  (let* ((alpha (vertex-pt pkt :face1 :bottom
                               :face2 :left
                           :face3 :front))
      (beta (vertex-pt pkt :face1 :top
                          :face2 :right
                          :face3 :back))
      (alpha (vector-to-list alpha))
      (beta (vector-to-list beta))
      (gamma (list (first alpha) (first beta)))
      (delta (list (second alpha) (+ (second alpha) 0.0001)))
      (zeta (list (third alpha) (third beta))))
    (get-islands (list gamma delta zeta) nil pkt (second beta))))


;;; Given the first interval from find-ribs-in-pocket, this
;;; function creates the pkt-island-list.
;;;
;;; input:  list1  This is a list of the intervals from which the
;;;                next geom will be created.
;;;         list2  This is the pkt-island-list being built.
;;;         pkt    The design pocket being translated.
```

```
;;;          max    The top of the pocket in the y direction.
;;;
;;; output: list2  The completed pkt-island-list.
;;;
(defun get-islands (list1 list2 pkt max)
  (let* ((pkt-geom (agm::rn-cube-geom list1))
      (list3 (find-ribs pkt-geom
                  (select :type 'rib-feature) nil)))

    (cond ((<= max (second (second list1))) list2)
       (t (get-islands (list (first list1)
                     (list (get-next-y max
                                 list3)
                           (+ (get-next-y max
                                    list3)
                              0.001))
                     (third list1))
                  (append (list
                        (list pkt
                           list3
                           (dietrick-round
                            (- (get-next-y max
                                     list3)
                               (first
                                (second list1))
                               )))))
                        list2)
                  pkt
                  max)))))


;;; This function will determine which ribs from a given list
;;; intersect the given thin plate geom.
;;;
;;; input:  feature   This is the thin plate geom.
;;;         list1     Is a list of ribs to be checked for
;;;                   intersection.
;;;
;;; output: list2     The list of intersecting ribs.
;;;
(defun find-ribs (feature list1 list2)
  (cond ((null list1) list2)

     ((agm::intersecting-geoms-p feature
                       (the geom (:from (car list1)))
                       0.00001)
      (find-ribs feature
              (cdr list1)
              (append (list (car list1)) list2)))

     (t (find-ribs feature (cdr list1) list2))
     ) ; end of conditional
  ) ; end of function
```

```
;;; This function determines the next smallest value of y (CM)
;;; and establishes the next y interval for the next thin plate.
;;;
;;; input:  n      This is the maximum value of y corresponding
;;;                to the top of the pocket.
;;;         list1  This is a list of the ribs detected in the last
;;;                thin plate investigation.
;;;
;;; output: A list of two numbers which is the y interval for
;;;         the next thin plate geom creation (see get-islands).
;;;
(defun get-next-y (n list1)
  (cond ((null list1) (+ n 0.0001))
     ((> n
         (second (vector-to-list (vertex-pt (car list1)
                                  :face1 :top
                                  :face2 :left
                                  :face3 :front))))
          (get-next-y (second (vector-to-list
                      (vertex-pt (car list1)
                                 :face1 :top
                                 :face2 :left
                                 :face3 :front)))
                (cdr list1)))
     (t (get-next-y n (cdr list1)))
     ) ; end of conditional
  ) ; end of function
```

APPENDIX B

MCAPP-FEATURES-LIST CODE

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;                                                  ;;;;;;;;
;;;;;;;;      FUNCTIONS TO DEFINE MCAPP-FEATURES-LIST     ;;;;;;;;
;;;;;;;;                                                  ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; This function repeatedly processes a single element of the
;;; pkt-island-list to build the mcapp-features-list property.
;;;
;;; input:  list1   The pkt-island-list which describes each
;;;         layer with a list.
;;;
;;; output: list2   A list of same layer lists composed of MetCAPP
;;;         features describing that layer.
;;;
(defun get-mcapp-features-list (list1 list2)
  (cond ((null list1) list2) ; output when done

     ((= (length (second (car list1))) 0) ; no islands
      (get-mcapp-features-list
       (cdr list1)
       (cons (list (get-mcapp-pocket
                 (first (car list1))
                 (third (car list1))))
          list2)))

     ((< (length (second (car list1))) 2) ; single island
      (get-mcapp-features-list
       (cdr list1)
       (cons (list (get-mcapp-features
                 (first (car list1))
                 (second (car list1))
                 (third (car list1))))
          list2)))

     (t (get-mcapp-features-list ; multiple islands
         (cdr list1)
         (cons (mult-make-mcapp-features
             (first  (car list1))
             (second (car list1))
```

```
            (third  (car list1)))
           list2)))

    ) ; end of conditional
  ) ; end of function


;;; This function develops the list of MetCAPP properties that
;;; define a MetCAPP pocket feature without any islands.
;;;
;;; input:  feature   This is the design feature being processed
;;;         by the positive features code.
;;;         height    This is the height of the layer which is
;;;         currently being processed.
;;;
;;; output: The list of MetCAPP properties defining a pocket.
;;;
(defun get-mcapp-pocket (feature height)
  (list
   (append (list (the d2-starting-block material
                 (:from (the d2-feature (:from feature))))
           (the metcapp-name
                 (:from (the d2-feature (:from feature))))
           (the machine
                 (:from (the d2-feature (:from feature))))
           (car (sort (list (the depth (:from feature))
                            (the width (:from feature)))
                      '>))
           (car (sort (list (the depth (:from feature))
                            (the width (:from feature)))
                      '<))
          height)
       (get-props feature
                 (the fillet-radius
                   (:from (the d2-feature
                              (:from feature)))
                   ))
       ) ;closes append
   )) ;closes list and function


;;; This function processes a single island interaction layer
;;;
;;; input:  feature   design feature being processed.
;;;         rib-list  list of ribs from the layer currently being
;;;                   processed.
;;;         height    height of layer currently being processed.
;;;
;;; output: A list of MetCAPP features defining the layer being
;;;         processed.
;;;
(defun get-mcapp-features (feature rib-list height)
  (let ((frbl-list (get-frbl-list feature (car rib-list)))
```

```
(cond ((null list1) list2)

   (t (mult-make-features
       feature
       blocks
       (cdr list1)
       height
       (cons
        (list
         (list (the d2-starting-block material
                (:from (the d2-feature (:from feature))))
               (the metcapp-name
                 (:from (the d2-feature (:from feature))))
               (the machine
                 (:from (the d2-feature (:from feature))))
               (first (sort (list
                       (get-x-dimension (car list1)
                                        blocks)
                       (get-z-dimension (car list1)
                                        blocks)) '>))
               (first (sort (list
                       (get-x-dimension (car list1)
                                        blocks)
                       (get-z-dimension (car list1)
                                        blocks)) '<))
               height
               (the corner-radius
                 (:from (the d2-feature (:from feature))))
               (the fillet-radius
                 (:from (the d2-feature (:from feature))))
               (the max-allow-cutter-dia
                 (:from (the d2-feature (:from feature))))
               (the max-height-obstruction
                 (:from (the d2-feature (:from feature))))
               (the thin-wall
                 (:from (the d2-feature (:from feature))))
               (the angle-floor-wall
                 (:from (the d2-feature (:from feature))))
               (the setup-rigidity
                 (:from (the d2-feature (:from feature))))
               nil ;;; sub-floor-length
               nil ;;; sub-floor-width
               nil ;;; sub-floor-height
               nil ;;; sub-floor-corner-radius
               nil ;;; sub-floor-fillet-radius
               ) ; end of metcapp parameters
         (car list1)
         blocks) ; end of second list
        list2)) ; end of list and cons
     ) ; end of condition
  ) ; end of conditional
) ; end of function
```

```
     ) ; end of declarations

    (cond ((or (eql (the fillet-radius
                (:from (the d2-feature (:from feature))))
               (the bottom-fillet-rad
                 (:from (car rib-list))))
           (eql 1 1)) ; To ensure this condition is followed
                      ; until the additional code is done.

        (get-four-regions feature
                    frbl-list
                    (the bottom-fillet-radius
                      (:from feature)) height))

       ; Code for these conditions not yet complete
       ((>   (the fillet-radius
               (:from (the d2-feature (:from feature))))
          (the bottom-fillet-rad (:from (car rib-list))))
        (append
         (get-four-regions feature
                     frbl-list
                     (the fillet-radius
                       (:from (the d2-feature
                                (:from feature))))
                     height)
         (finish-feature feature
                    (car rib-list)
                    frbl-list
                    (the bottom-fillet-rad
                      (:from (car rib-list)))))
        ) ; end of condition

       (t (append
           (get-four-regions feature
                     frbl-list
                     (the bottom-fillet-rad
                          (:from (car rib-list)))
                     height)
           (finish-feature feature
                     (car rib-list)
                     frbl-list
                     (the fillet-radius
                       (:from (the d2-feature
                                (:from feature)))))
           )) ; closes append and condition
       ) ; end of conditional
    )) ; end of if, let, and function


;;; This function determines the front, right, back, and left
;;; dimensions when a single island intersects a pocket layer.
;;;
;;; input: feature   The design feature being processed by the
```

```
;;;                    positive features code.
;;;         rib        The rib feature intersecting the layer of the
;;;                    feature currently being processed.
;;;
;;; output: A list of the front, right, back, and left.
;;;
(defun get-frbl-list (feature rib)
  (list (vector-dot-product
      (get-vector 0 0 1)
      (subtract-points :point1 (vertex-pt feature
                               :face1 :bottom
                               :face2 :left
                               :face3 :front)
                  :point2 (vertex-pt rib
                               :face1 :bottom
                               :face2 :left
                               :face3 :front)))
     (vector-dot-product
      (get-vector 1 0 0)
      (subtract-points :point1 (vertex-pt feature
                               :face1 :bottom
                               :face2 :right
                               :face3 :back)
                  :point2 (vertex-pt rib
                               :face1 :bottom
                               :face2 :right
                               :face3 :back)))
     (vector-dot-product
      (get-vector 0 0 1)
      (subtract-points :point1 (vertex-pt rib
                               :face1 :bottom
                               :face2 :right
                               :face3 :back)
                  :point2 (vertex-pt feature
                               :face1 :bottom
                               :face2 :right
                               :face3 :back)))
     (vector-dot-product
      (get-vector 1 0 0)
      (subtract-points :point1 (vertex-pt rib
                               :face1 :bottom
                               :face2 :left
                               :face3 :front)
                  :point2 (vertex-pt feature
                               :face1 :bottom
                               :face2 :left
                               :face3 :front)))
     )) ; closes list and function


;;; This function actually creates the MetCAPP properties that
;;; define a four MetCAPP features that represent a single layer
;;; of an interaction with one island.
```

```
;;;
;;; input:  feature        Feature being processed.
;;;         frbl-list      List of dimensions.
;;;         fillet-radius  To be used.
;;;         height         Of the layer being processed.
;;;
;;; output: The MetCAPP properties defining four features.
;;;
(defun get-four-regions (feature frbl-list fillet-radius height)
      (list (append (list (the d2-starting-block material
                        (:from (the d2-feature
                              (:from feature))))
                  (the metcapp-name
                        (:from (the d2-feature
                              (:from feature))))
                  (the machine
                        (:from (the d2-feature
                              (:from feature))))
                  (car (sort (list (first frbl-list)
                                   (the width
                                     (:from feature)))
                             '>))
                  (car (sort (list (first frbl-list)
                                   (the width
                                     (:from feature)))
                             '<))
                  height)
            (get-props feature fillet-radius))

        (append (list (the d2-starting-block material
                        (:from (the d2-feature
                              (:from feature))))
                  (the metcapp-name
                        (:from (the d2-feature
                              (:from feature))))
                  (the machine
                        (:from (the d2-feature
                              (:from feature))))
                  (car (sort (list (the depth
                                     (:from feature))
                                   (second frbl-list))
                             '>))
                  (car (sort (list (the depth
                                     (:from feature))
                                   (second frbl-list))
                             '<))
                  height)
            (get-props feature fillet-radius))

        (append (list (the d2-starting-block material
                        (:from (the d2-feature
                              (:from feature))))
                  (the metcapp-name
```

```
                    (:from (the d2-feature
                           (:from feature))))
               (the machine
                    (:from (the d2-feature
                           (:from feature))))
               (car (sort (list (third frbl-list)
                           (the width
                             (:from feature)))
                      '>))
               (car (sort (list (third frbl-list)
                           (the width
                             (:from feature)))
                      '<))
               height)
          (get-props feature fillet-radius))

     (append (list (the d2-starting-block material
                    (:from (the d2-feature
                           (:from feature))))
               (the metcapp-name
                    (:from (the d2-feature
                           (:from feature))))
               (the machine
                    (:from (the d2-feature
                           (:from feature))))
               (car (sort (list (the depth
                             (:from feature))
                           (fourth frbl-list))
                      '>))
               (car (sort (list (the depth
                             (:from feature))
                           (fourth frbl-list))
                      '<))
               height)
          (get-props feature fillet-radius))))


;;; Future function referenced by get-mcapp-features but not
;;; yet complete.
;;;
(defun finish-feature (pocket rib frbl-list fillet-radius)
  'need-code-here)


;;; This function defines a fraction of the MetCAPP properties.
;;;
;;; input:  feature        Design feature being processed.
;;;         fillet-radius  To be used.
;;;
;;; output: A list of some of the MetCAPP properties.
;;;
(defun get-props (feature fillet-radius)
```

```
  (list (the corner-radius
          (:from (the d2-feature (:from feature))))
     fillet-radius
     (the max-allow-cutter-dia
          (:from (the d2-feature (:from feature))))
     (the max-height-obstruction
          (:from (the d2-feature (:from feature))))
     (the thin-wall
          (:from (the d2-feature (:from feature))))
     (the thin-floor
          (:from (the d2-feature (:from feature))))
     (the angle-floor-wall
          (:from (the d2-feature (:from feature))))
     (the setup-rigidity
          (:from (the d2-feature (:from feature))))
     nil  ;;; sub-floor-length
     nil  ;;; sub-floor-width
     nil  ;;; sub-floor-height
     nil  ;;; sub-floor-corner-radius
     nil  ;;; sub-floor-fillet-radius
     ))


;;; This function determines the MetCAPP features for a single
;;; layer where multiple islands are involved.
;;;
;;; input:  feature      Negative feature being processed.
;;;         rib-list     List of ribs at the given layer.
;;;         height       Of the layer being processed.
;;;
;;; output: A list of MetCAPP features for the processed layer.
;;;
(defun mult-make-mcapp-features (feature rib-list height)
  (let* ((x-values (remove-duplicates
             (sort (get-x-values feature
                         rib-list
                         nil) '<)))
      (z-values (remove-duplicates
             (sort (get-z-values feature
                         rib-list
                         nil) '<)))
      (nodes (make-nodes x-values
                  z-values
                  0
                  0
                  (length x-values)
                  (length z-values)
                  nil))
      (grid (make-blocks nodes
                  0
                  0
                  (- (length x-values) 2)
                  (- (length z-values) 2)
```

```
                nil))
      (blocks (check-grid grid rib-list nil))
      (regions (mult-make-regions blocks nil))
      ) ; end of variable definitions

    (mult-make-features feature blocks regions height nil)

    ) ; end of let statement
  ) ; end of function


;;; This function will determine the x coordinates for the nodes
;;; of the non-uniform grid and assemble them in a list.
;;;
;;; input:  feature   Refers to the negative feature under
;;;                   examination.
;;;         list1     Refers to the list of positive features
;;;                   impacting the above mentioned feature.
;;;
;;; output: list2     The list of x coordinates for the grid.
;;;
(defun get-x-values (feature list1 list2)
  (cond ((null list1)
      (append (list (first (vector-to-list
                        (vertex-pt feature
                             :face1 :bottom
                             :face2 :left
                             :face3 :back)))
                  (first (vector-to-list
                        (vertex-pt feature
                             :face1 :bottom
                             :face2 :right
                             :face3 :front))))
          list2)
      ) ; end of condition

     (t (get-x-values
         feature
         (cdr list1)
         (append (list (first (vector-to-list
                        (vertex-pt (car list1)
                                :face1 :bottom
                                :face2 :left
                                :face3 :back)))
                  (first (vector-to-list
                        (vertex-pt (car list1)
                                :face1 :bottom
                                :face2 :right
                                :face3 :front)))
                  ) ; closes list

             list2)  ; closes append
        ))              ; closes condition
```

```
    ))                  ; closes conditional & function

;;; This function will determine the z coordinates for the nodes
;;; of the non-uniform grid and assemble them in a list.
;;;
;;; input:  feature   Refers to the negative feature under
;;;                   examination.
;;;         list1     Refers to the list of positive features
;;;                   impacting the above mentioned feature.
;;;
;;; output: list2     The list of z coordinates for the grid.
;;;
(defun get-z-values (feature list1 list2)
  (cond ((null list1)
      (append (list (third (vector-to-list
                        (vertex-pt feature
                             :face1 :bottom
                             :face2 :left
                             :face3 :back)))
                    (third (vector-to-list
                        (vertex-pt feature
                             :face1 :bottom
                             :face2 :right
                             :face3 :front))))
          list2)
    )

    (t (get-z-values
        feature
        (cdr list1)
        (append (list (third (vector-to-list
                        (vertex-pt (car list1)
                                :face1 :bottom
                                :face2 :left
                                :face3 :back)))
                    (third (vector-to-list
                        (vertex-pt (car list1)
                                :face1 :bottom
                                :face2 :right
                                :face3 :front)))
                    ) ; closes list

                list2)  ; closes append
        ))              ; closes second argument
    ))                  ; closes conditional and function


;;; This function will create the nodes of the non-uniform grid
;;; that is imposed on a layer of an interaction involving
;;; multiple islands.
;;;
;;; input:  list1   Is a list of CM x-values for the grid.
```

```
;;;          list2    Is a list of CM z-values for the grid.
;;;          x        Is a counter for use in an nth-function to
;;;                   extract specific elements from list1.
;;;          z        Is a counter for use in an nth-function to
;;;                   extract specific elements from list2.
;;;          x-max    Is the original length of list1 with the first
;;;                   element counted as 1.
;;;          z-max    Is the original length of list2 with the first
;;;                   element counted as 1.
;;;
;;; output: list3    Initially set to nil, list3 is the list of
;;;                   nodes defining the grid.
;;;
(defun make-nodes (list1 list2 x z x-max z-max list3)
  (cond ((>= z z-max) (reverse list3))
     ((>= x x-max)
      (make-nodes list1 list2 1
                (+ z 1) x-max z-max
                (if (= (+ z 1) z-max)
                 (cons (reverse (car list3))
                       (cdr list3))
                  (cons (list (list (first list1)
                                    (nth (+ z 1) list2)))
                        (cons (reverse (car list3))
                              (cdr list3)))
                 ) ; endif
                )  ; end of call to make-nodes
      ) ; end of second argument of conditional

     (t (make-nodes list1 list2 (+ x 1) z x-max z-max
                 (cons (cons (list (nth x list1)
                                   (nth z list2))
                             (car list3))
                       (cdr list3))))
     ) ; end of conditional
  ) ; end of function


;;; This function transforms the node list into the grid.
;;;
;;; input:  list1    Is the node list.
;;;         x and z  Are array counters initially set to 0.
;;;         x-max    Is the length of the internal list of nodes
;;;                  (the number of columns) beginning with zero.
;;;         z-max    Is the length of the external list of lists
;;;                  (the number of rows) beginning with zero.
;;;
;;; output: list2    Initially set to nil, list2 is the grid.
;;;
(defun make-blocks (list1 x z x-max z-max list2)
  (cond ((> z z-max)
      (reverse list2))
```

```
      ((> x x-max)
       (make-blocks list1 0 (+ z 1) x-max z-max
                 (cons (reverse (car list2))
                    (cdr list2))))

      ((= x 0)
       (make-blocks list1 (+ x 1) z x-max z-max
                 (cons
                  (list
                 (list (nth x (nth z list1))
                       (nth x (nth (+ z 1) list1))
                       (nth (+ x 1) (nth z list1))
                       (nth (+ x 1) (nth (+ z 1) list1)))
                 ) ; end of first list
                    list2)))

      (t (make-blocks list1 (+ x 1) z x-max z-max
                 (cons
                  (cons
                   (list (nth x (nth z list1))
                      (nth x (nth (+ z 1) list1))
                      (nth (+ x 1) (nth z list1))
                      (nth (+ x 1) (nth (+ z 1) list1)))
                   (car list2))
                  (cdr list2))))
      ) ; end of conditional
    ) ; end of function


;;; This function will examine the grid and note where positive
;;; features exist on the grid.
;;;
;;; input:  list1     This is the list of blocks.
;;;         rib-list  List of ribs at the given layer.
;;;
;;; output: list2     Initially set to nil, this is the
;;;                   processed grid.
;;;
(defun check-grid (list1 rib-list list2)
  (cond ((null list1) (reverse list2))

      (t (check-grid (cdr list1)
                  rib-list
                  (cons (check-grid-row (car list1)
                                        rib-list
                                        nil)
                     list2)
                  ))
      ) ; end of conditional
    ) ; end of function


;;; This function will check a single row of the grid for
```

```
;;; positive features.
;;;
;;; input:  list1     This is a list describing a row of the grid.
;;;         rib-list  A list of ribs for the given layer.
;;;
;;; output: list2     Initially set to nil, this is the
;;;                   processed row of the grid.
;;;
(defun check-grid-row (list1 rib-list list2)
  (cond ((null list1) (reverse list2))

     (t (check-grid-row (cdr list1)
                  rib-list
                  (cons (check-for-block rib-list
                                         (car list1))
                     list2)
                  ))
     ) ; end of conditional
  ) ; end of function


;;; This function will check a single rectangle for a positive
;;; feature using the agm solid modeller.
;;;
;;; input:  list1   This is a list of ribs that will be checked.
;;;         grid    This is the single rectangle that will be
;;;                 checked.
;;;
;;; output: If there is an intersection, 'block will be returned.
;;;         If not, the rectangle (grid) will be returned.
;;;
(defun check-for-block (list1 grid)
  (let* ((y (second (vector-to-list (vertex-pt (car list1)
                                    :face1 :bottom
                                    :face2 :left
                                    :face3 :back))))
      (cube (agm::rn-cube-geom
          (list (list (+ (first (first grid))
                         0.0001)
                      (- (first (third grid))
                         0.0001))
                (list y
                   (+ y 0.1))
                (list (+ (second (first grid))
                         0.0001)
                      (- (second (second grid))
                         0.0001))
                ))
            ) ; end of cube definition
      ) ; end of variable definitions

    (cond ((agm::intersecting-geoms-p (the geom
                                (:from (car list1)))
```

```
                        cube
                        0.00001)
        '(block))

       (t
        (if (null (cdr list1))
            grid
          (check-for-block (cdr list1) grid)))

       ) ; end of conditional
    ) ; end of let
  ) ; end of function


;;; This function controls the functions to repeatedly link
;;; rectangles of the grid into MetCAPP features.
;;;
;;; input:  blocks  This is the list which describes the grid.
;;;
;;; output: list1   Initially set to nil, this is a list of the
;;;                 created MetCAPP features in terms of
;;;                 rectangles.
;;;
(defun mult-make-regions (blocks list1)
  (let ((home (get-next-home blocks (join-lists list1 nil) 0 0))
     ) ; end of variable definitions

    (cond ((equal home 'done) list1)

       (t (mult-make-regions
           blocks
           (cons (join-blocks t t t t blocks home)
              list1))
          ) ; end of condition
       ) ; end of conditional
    ) ; end of let statement
  ) ; end of function


;;; This function is used to determine the "home" of the next
;;; MetCAPP feature to be created. The home refers to the first
;;; and possibly the only rectangle to compose a MetCAPP feature.
;;; If a new home does not exist, meaning that the entire layer
;;; has been mapped into MetCAPP features, 'done is returned.
;;;
;;; input:  blocks   This is the list representing the grid.
;;;         list1    This is the list of MetCAPP features already
;;;                  created.
;;;         row      An index for the row.
;;;         column   An index for the column.
;;;
;;; output: The home for the next MetCAPP feature.
;;;
```

```
(defun get-next-home (blocks list1 row column)
  (let ((home (nth column (nth row blocks)))
     ) ; end of variable definitions

    (cond ((null home)
        (if (eql (length blocks) row)
            'done
          (get-next-home blocks list1 (+ row 1) 0)
          ) ; endif
        ) ; end of condition

       ((equal home '(block))
        (get-next-home blocks list1 row (+ column 1))
        ) ; end of condition

       ((bob-subsetp (list (list row column)) list1)
        (get-next-home blocks list1 row (+ column 1))
        ) ; end of condition

       (t (list (list row column)))
       ) ; end of conditional
    ) ; end of let statement
  ) ; end of function


;;; This function will attempt to join additional rectangles to
;;; the home of a given MetCAPP feature.
;;;
;;; input:  +x      This is the key to determine if expansion in
;;;                 the +x direction is possible.
;;;         +z      This is the key to determine if expansion in
;;;                 the +z direction is possible.
;;;         -x      This is the key to determine if expansion in
;;;                 the -x direction is possible.
;;;         -z      This is the key to determine if expansion in
;;;                 the -z direction is possible.
;;;                 For all of the above, t indicates possible,
;;;                 and nil indicates impossible.
;;;         blocks  This is the representation of the grid.
;;;
;;; output: list1   Is an index list to determine which blocks
;;;                 have been joined.
;;;
(defun join-blocks (+x +z -x -z blocks list1)
  (let ((x (get-x-dimension list1 blocks))
     (z (get-z-dimension list1 blocks))
     ) ; end of variable definitions

    (cond ((and (<= z x) +z)
        (if (eql (expand+z list1 blocks) 'block)
            (join-blocks +x nil -x -z blocks list1)
          (join-blocks +x +z -x -z blocks
                (append (expand+z list1 blocks)
```

```
                    list1))
      ) ; endif
     ) ; end of condition

    (+x
     (if (eql (expand+x list1 blocks) 'block)
         (join-blocks nil +z -x -z blocks list1)
       (join-blocks +x +z -x -z blocks
              (append (expand+x list1 blocks)
                    list1))
      ) ; endif
     ) ; end of condition

    (+z
     (if (eql (expand+z list1 blocks) 'block)
         (join-blocks +x nil -x -z blocks list1)
       (join-blocks +x +z -x -z blocks
              (append (expand+z list1 blocks)
                    list1))
      ) ; endif
     ) ; end of condition

    ((and (<= z x) -z)
     (if (eql (expand-z list1 blocks) 'block)
         (join-blocks +x +z -x nil blocks list1)
       (join-blocks +x +z -x -z blocks
              (append (expand-z list1 blocks)
                    list1))
      ) ; endif
     ) ; end of condition

    (-x
     (if (eql (expand-x list1 blocks) 'block)
         (join-blocks +x +z nil -z blocks list1)
       (join-blocks +x +z -x -z blocks
              (append (expand-x list1 blocks)
                    list1))
      ) ; endif
     ) ; end of condition

    (-z
     (if (eql (expand-z list1 blocks) 'block)
         (join-blocks +x +z -x nil blocks list1)
       (join-blocks +x +z -x -z blocks
              (append (expand-z list1 blocks)
                    list1))
      ) ; endif
     ) ; end of condition

    (t list1)

   ) ; end of conditional
) ; end of let statement
```

```
  ) ; end of function


;;; This function will attempt to expand the MetCAPP feature in
;;; the +x direction.
;;;
;;; input:  list1    This is the list representing the current
;;;                  MetCAPP feature.
;;;         blocks   This is the list representing the grid.
;;;
;;; output: list1    In the process of expansion, list1 is
;;;                  updated to show the increased feature.
;;;
(defun expand+x (list1 blocks)
  (let ((x (car (sort
           (get-all-nth 1 list1 nil)
           '>)))
     (z-list (remove-duplicates
           (sort (get-all-nth 0 list1 nil) '<))
         ) ; end of z-list definition

     ) ; end of variable definitions

    (expand-bob 'x (+ x 1) z-list blocks nil)
    ) ; end of let
  ) ; end of function


;;; This function will attempt to expand the MetCAPP feature in
;;; the +z direction.
;;;
;;; input:  list1    This is the list representing the current
;;;                  MetCAPP feature.
;;;         blocks   This is the list representing the grid.
;;;
;;; output: list1    In the process of expansion, list1 is
;;;                  updated to show the increased feature.
;;;
(defun expand+z (list1 blocks)
  (let ((z (car (sort
           (get-all-nth 0 list1 nil)
           '>)))
     (x-list (remove-duplicates
           (sort (get-all-nth 1 list1 nil) '<))
         ) ; end of x-list definition

     ) ; end of variable definitions

    (expand-bob 'z (+ z 1) x-list blocks nil)
    ) ; end of let
  ) ; end of function
```

```
;;; This function will attempt to expand the MetCAPP feature in
;;; the -x direction.
;;;
;;; input:  list1    This is the list representing the current
;;;                  MetCAPP feature.
;;;         blocks   This is the list representing the grid.
;;;
;;; output: list1    In the process of expansion, list1 is
;;;                  updated to show the increased feature.
;;;
(defun expand-x (list1 blocks)
  (let ((x (car (sort
           (get-all-nth 1 list1 nil)
           '<)))
     (z-list (remove-duplicates
           (sort (get-all-nth 0 list1 nil) '<))
         ) ; end of z-list definition

     ) ; end of variable definitions

    (if (< (- x 1) 0)
     'block
      (expand-bob 'x (- x 1) z-list blocks nil)
      ) ; endif
    ) ; end of let
  ) ; end of function


;;; This function will attempt to expand the MetCAPP feature in
;;; the -z direction.
;;;
;;; input:  list1    This is the list representing the current
;;;                  MetCAPP feature.
;;;         blocks   This is the list representing the grid.
;;;
;;; output: list1    In the process of expansion, list1 is
;;;                  updated to show the increased feature.
;;;
(defun expand-z (list1 blocks)
  (let ((z (car (sort
           (get-all-nth 0 list1 nil)
           '<)))
     (x-list (remove-duplicates
           (sort (get-all-nth 1 list1 nil) '<))
         ) ; end of x-list definition

     ) ; end of variable definitions

    (if (< (- z 1) 0)
     'block
      (expand-bob 'z (- z 1) x-list blocks nil)
      ) ; endif
    ) ; end of let
```

```
  ) ; end of function


;;; This is the generic function to actually do the expansion.
;;;
;;; input:  x-or-z   Identifies expansion into the x or z
;;;                  direction.
;;;         n        Is the row or column to expand into.
;;;         list1    Is a list of columns or rows to expand over.
;;;         blocks   This is the grid representation.
;;;
;;; output: list2    Initially set to nil, this list will be
;;;                  the rectangles included by the expansion.
;;;
(defun expand-bob (x-or-z n list1 blocks list2)
  (cond ((null list1) list2)

     ((eql x-or-z 'x)
      (if (or (equal (nth n (nth (car list1) blocks))
                     '(block))
              (null (nth n (nth (car list1) blocks))))
          'block
        (expand-bob x-or-z n (cdr list1) blocks
                    (cons (list (car list1) n)
                          list2))
        ) ; endif
      )   ; end of second condition

     (t (if (or (equal (nth (car list1) (nth n blocks))
                       '(block))
                (null (nth (car list1) (nth n blocks))))
            'block
          (expand-bob x-or-z n (cdr list1) blocks
                      (cons (list n (car list1))
                            list2))
          ) ; endif
        )   ; end of condition
     ) ; end of conditional
  ) ; end of function


;;; This function determines the MetCAPP features for a given
;;; layer based on input from mult-make-mcapp-features.
;;;
;;; input:  feature  The design feature being processed.
;;;         blocks   This list represents the grid rectangles.
;;;         list1    Is an index to blocks list.
;;;         height   Of the layer being processed.
;;;
;;; output: list2    Initially set to nil, list2 is the list of
;;;                  the MetCAPP features for that layer.
;;;
(defun mult-make-features (feature blocks list1 height list2)
```

APPENDIX C

POSITIVE-OPS CODE

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;                                                    ;;;;;;;;
;;;;;;;;    FUNCTIONS TO DEFINE POSITIVE-OPS PROPERTY       ;;;;;;;;
;;;;;;;;                                                    ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; This function repeatedly sends a single element of list1 to
;;; get-mcapp-prime.
;;;
;;; input:  list1   The mcapp-features-list from a given feature.
;;;         It is a list of lists of same layer MetCAPP features
;;;         that completely define the positive-negative feature
;;;         interaction.
;;;
;;; output: list2   Initially set to nil, list2 becomes the
;;;         positive-ops property of a feature. It is a list of
;;;         lists of same layer operations that completely define
;;;         the positive-negative feature interaction.
;;;
(defun get-metcapp-ops (list1 list2)
  (cond ((null list1) list2)
     ( t (get-metcapp-ops
          (cdr list1)
          (append (list (get-mcapp-prime (car list1) nil))
               list2)))
     ))

;;; This function repeatedly sends a single element of list1 to
;;; get-mcapp.
;;;
;;; input:  list1   A list of lists of MetCAPP properties
;;;         completely describing a single layer of interaction
;;;
;;; output: list2   Initially set to nil, list2 becomes the
;;;         list of lists of operations returned from MetCAPP.
;;;
(defun get-mcapp-prime (list1 list2)
  (cond ((null list1) list2)
     ( t (get-mcapp-prime
          (cdr list1)
          (append (list (get-mcapp (car (car list1))))
```

```
              list2)))
    ))

;;; This function calls MetCAPP with a single list of properties
;;; that define a given MetCAPP feature.
;;;
;;; input:  list1   A list of MetCAPP properties
;;;
;;; output: A list of operations from MetCAPP
;;;
(defun get-mcapp (list1)
  (let* ((save-pointer
       (lcl:make-foreign-pointer
        :address (man-api                ;call to MetCapp
                 (set-ip in-pointer    ;convert to strings
                      (mapcar 'prop-to-string list1)))
        :type '(:pointer output)))
      (save-operations
       (many-structs-to-lists     ;convert C struct to list
        save-pointer)))
    (free-operations-memory save-pointer)
    save-operations)
  )
```

APPENDIX D

ROUGH-TOOL-LIST CODE

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;                                                           ;;;;;;;;
;;;;;;;;     FUNCTIONS TO DEFINE THE ROUGH-TOOL-LIST               ;;;;;;;;
;;;;;;;;                                                           ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; This function is responsible for building the rough-tool-list
;;; by repeatedly sending a list of the operations for each layer
;;; of the interaction to the get-rough-tools function.
;;;
;;; input:  list1  The positive-ops list for a feature which is
;;;                a list of operations by layer and then by
;;;                MetCAPP feature.
;;;
;;; output: list2  The completed rough-tool-list which is a list
;;;                of the rough tools required for that
;;;                interaction by layer.
;;;
(defun get-rough-tool-list (list1 list2)
  (cond ((null list1) (reverse list2))
     ( t (get-rough-tool-list (cdr list1)
                    (cons (get-rough-tools
                        (car list1)
                        nil)
                          list2)))
     ) ; end of conditional
  ) ; end of function


;;; This function is responsible for extracting the appropriate
;;; roughing tools for a layer from a list of the operations for
;;; all of the MetCAPP features of that layer.
;;;
;;; input:  list1   List of the operations by MetCAPP feature for
;;;                 the layer being processed.
;;;
;;; output: list2   Initially set to nil, list2 is the completed
;;;                 list of roughing tools for that layer.
;;;
```

```
(defun get-rough-tools (list1 list2)
  (cond ((null list1) list2)
     ( t (get-rough-tools (cdr list1)
                    (append (list
                          (third
                           (third (first list1))))
                         list2)))
     ) ; end of conditional
  ) ; end of function
```

APPENDIX E

FINISH-TOOL-LIST CODE

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;                                                 ;;;;;;;;
;;;;;;;;     FUNCTIONS TO DEFINE THE FINISH-TOOL-LIST    ;;;;;;;;
;;;;;;;;                                                 ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; This function is responsible for building the finish-tool-list
;;; by repeatedly sending a list of the operations for each layer
;;; of the interaction to the get-finish-tools function.
;;;
;;; input:  list1  The positive-ops list for a feature which is
;;;                a list of operations by layer and then by
;;;                MetCAPP feature.
;;;
;;; output: list2  The completed finish-tool-list which is a list
;;;                of the rough tools required for that
;;;                interaction by layer.
;;;
(defun get-finish-tool-list (list1 list2)
  (cond ((null list1) (reverse list2))
     ( t (get-finish-tool-list (cdr list1)
                  (cons (get-finish-tools
                      (car list1)
                      nil)
                     list2)))
     ) ; end of conditional
  ) ; end of function


;;; This function is responsible for extracting the appropriate
;;; finishing tools for a layer from a list of the operations for
;;; all of the MetCAPP features of that layer.
;;;
;;; input:  list1   List of the operations by MetCAPP feature for
;;;                 the layer being processed.
;;;
;;; output: list2   Initially set to nil, list2 is the completed
;;;                 list of finishing tools for that layer.
;;;
```

```
(defun get-finish-tools (list1 list2)
  (cond ((null list1) list2)
     ( t (get-finish-tools (cdr list1)
                    (cons (third
                        (car
                         (last (first list1)))
                        ) ; closes third
                       list2)))
     ) ; end of conditional
  ) ; end of function
```

APPENDIX F

MACHINING REGIONS CODE

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;                                                      ;;;;;;;;
;;;;;;;;    FUNCTIONS TO DEFINE THE MACHINING-REGIONS         ;;;;;;;;
;;;;;;;;                                                      ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;; This function is responsible for determining the machining
;;; regions layer by layer for an interaction.
;;;
;;; input:  list1  Is the pkt-islnad-list from the d2-pocket being
;;;                processed.
;;;         list2  Is the rough-tool-list from the pocket.
;;;         list4  Is (the metcapp-features-list (:from pocket)).
;;;
;;; output: list3  The machining-regions (rough or finish) for
;;;                the processed pocket.
;;;;;; This function is responsible for determining the machining
;;; regions layer by layer for an interaction.
;;;
;;; input:  list1  Is the pkt-islnad-list from the d2-pocket being
;;;                processed.
;;;         list2  Is the rough-tool-list from the pocket.
;;;         list4  Is (the metcapp-features-list (:from pocket)).
;;;
;;; output: list3  The machining-regions (rough or finish) for
;;;                the processed pocket.
;;;
(defun get-machining-regions (list1 list2 list3 list4)
  (cond ((null list1) (reverse list3))

     ((eql (length (second (car list1))) 0) ; no islands
      (get-machining-regions
       (cdr list1)
       (cdr list2)
       (cons (list (get-pocket-region (first (car list1))
                                      (car list2))
                  (third (car list1)))
          list3)
       (cdr list4)))
```

```
     ((eql (length (second (car list1))) 1) ; one island
      (get-machining-regions
       (cdr list1)
       (cdr list2)
       (cons (list (get-regions (first (car list1))
                                (car (second (car list1)))
                                (car list2))
                   (third (car list1)))
             list3)
       (cdr list4)))

     ((> (length (second (car list1))) 1) ; multiple islands
      (get-machining-regions
       (cdr list1)
       (cdr list2)
       (cons (get-mult-machining-regions (car list4)
                                         (car list2))
          list3)
       (cdr list4)))
     ) ; end of conditional
  ) ; end of function


;;; This function will establish the required information for NC
;;; code generation for a layer without any islands of a
;;; rectangular pocket region.
;;;
;;; input:  pocket     The pocket feature being processed.
;;;         tool-list  The tool-list (rough or finish) for that
;;;                    layer.
;;;
;;; output: (((apt-vertices) tool-id 'pocket))
;;;
(defun get-pocket-region (pocket tool-list)
  (list (list (get-apt-coords (vertex-pt pocket
                              :face1 :bottom
                              :face2 :left
                              :face3 :front))
              (get-apt-coords (vertex-pt pocket
                              :face1 :bottom
                              :face2 :right
                              :face3 :front))
              (get-apt-coords (vertex-pt pocket
                              :face1 :bottom
                              :face2 :right
                              :face3 :back))
              (get-apt-coords (vertex-pt pocket
                              :face1 :bottom
                              :face2 :left
                              :face3 :back)))
     (car tool-list)
     'pocket))
```

```
;;; This function groups the regions for a single island.
;;;
(defun get-regions (pocket rib tool-list)
;;;
;;; Points A through P are the vertices of the pocket and the rib
;;; and the projection of the rib's vertices onto the pocket.
;;; See the diagram below:
;;;
;;;        J         I       H              G
;;;
;;;
;;;
;;;        K         P       O              F
;;;
;;;
;;;
;;;        L         M       N -            E
;;;
;;;
;;;
;;;        A         B       C              D
;;;
;;;  Points A D G and J are the four corners of the pocket.
;;;  Points M N O and P are the four corners of the rib.
;;;  Points B C E F H I K and L are the projection of points
;;;  M N O and P onto each side of the pocket.
;;;
  (let* ((tool-1 (first  tool-list))
      (tool-2 (second tool-list))
      (tool-3 (third  tool-list))
      (tool-4 (fourth tool-list))
      (A (vertex-pt pocket :face1 :bottom
                           :face2 :front
                    :face3 :left))
      (D (vertex-pt pocket :face1 :bottom
                           :face2 :front
                    :face3 :right))
      (G (vertex-pt pocket :face1 :bottom
                           :face2 :back
                    :face3 :right))
      (J (vertex-pt pocket :face1 :bottom
                           :face2 :back
                    :face3 :left))
      (M (get-apt-coords (vertex-pt rib :face1 :bottom
                                        :face2 :front
                                 :face3 :left)))
      (N (get-apt-coords (vertex-pt rib :face1 :bottom
                                 :face2 :front
                                 :face3 :right)))
      (O (get-apt-coords (vertex-pt rib :face1 :bottom
                                 :face2 :back
                                 :face3 :right)))
      (P (get-apt-coords (vertex-pt rib :face1 :bottom
```

```
                        :face2 :back
                        :face3 :left)))
      (front (first  (get-frbl-list pocket rib)))
      (right (second (get-frbl-list pocket rib)))
      (back  (third  (get-frbl-list pocket rib)))
      (B (get-apt-coords
          (add-points :point1 A
                :point2 (get-vector left 0 0))))
      (C (get-apt-coords
          (subtract-points :point1 D
                      :point2 (get-vector right 0 0))))
      (I (get-apt-coords
          (add-points :point1 J
                :point2 (get-vector left 0 0))))
      (H (get-apt-coords
          (subtract-points :point1 G
                      :point2 (get-vector right 0 0))))
      (F (get-apt-coords
          (add-points :point1 G
                :point2 (get-vector 0 0 back))))
      (E (get-apt-coords
          (subtract-points :point1 D
                      :point2 (get-vector 0 0 front))))
      (K (get-apt-coords
          (add-points :point1 J
                :point2 (get-vector 0 0 back))))
      (L (get-apt-coords
          (subtract-points :point1 A
                      :point2 (get-vector 0 0 front))))

      (A (get-apt-coords A))
      (D (get-apt-coords D))
      (G (get-apt-coords G))
      (J (get-apt-coords J))


;;;
;;;
;;;
      (pocket-1 (if (and (< left front) (< right front))
                    (list (list A D E L) tool-1 'enclosed-pocket)
                  (if (< left front)
                    (list (list A C N L) tool-1 'open-pocket)
                      (if (< right front)
                        (list (list B D E M) tool-1 'open-pocket)
                      (list (list B C N M) tool-1 'slot)))))
;;;
;;;
;;;
      (pocket-3 (if (and (< left back) (< right back))
                    (list (list K F G J) tool-3 'enclosed-pocket)
                  (if (< left back)
                    (list (list K O H J) tool-3 'open-pocket)
```

```
                (if (< right back)
                  (list (list P F G I) tool-3 'open-pocket)
                (list (list P O H I) tool-3 'slot)))))

;;;
;;;
;;;
      (pocket-2 (if (and (< back right) (< front right))
                  (list (list C D G H) tool-2 'enclosed-pocket)
                (if (< front right)
                  (list (list C D F O) tool-2 'open-pocket)
                  (if (< back right)
                    (list (list N E F G H) tool-2 'open-pocket)
                  (list (list N E F O) tool-2 'slot)))))
;;;
;;;
;;;
      (pocket-4 (if (and (< back left) (< front left))
                  (list (list A B I J) tool-4 'enclosed-pocket)
                (if (< front left)
                  (list (list A B P K) tool-4 'open-pocket)
                  (if (< back left)
                    (list (list L M I J) tool-4 'open-pocket)
                  (list (list L M P K) tool-4 'slot))))))

(cond ((and (equal tool-1 tool-2) (equal tool-2 tool-3) (equal tool-3
tool-4))
        (list (list (list A D G J) (list M N O P)) tool-1
'O-pocket))

       ((and (equal tool-1 tool-2) (equal tool-2 tool-3))
        (if (< left right)
            (list (list (list A D G J K O N L) tool-1
'U-enclosed-pocket)
                  pocket-4)
          (list (list (list B D G I P O N M) tool-1 'U-slot)
            pocket-4)))

       ((and (equal tool-2 tool-3) (equal tool-3 tool-4))
        (if (< front back)
            (list (list (list A B P O C D G J) tool-2
'U-enclosed-pocket)
                  pocket-1)
          (list (list (list L M P O N E G J) tool-2 'U-slot)
            pocket-1)))

       ((and (equal tool-3 tool-4) (equal tool-4 tool-1))
        (if (< right left)
            (list (list (list A D E M P F G J) tool-3
'U-enclosed-pocket)
                  pocket-2)
          (list (list (list A C N M P O H J) tool-3 'U-slot)
            pocket-2)))
```

```
((and (equal tool-4 tool-1) (equal tool-1 tool-2))
 (if (< back front)
     (list  (list  (list  A  D  G  H  N  M  I  J)  tool-4
'U-enclosed-pocket)
           pocket-3)
   (list (list (list A D F O N M P K) tool-4 'U-slot)
      pocket-3)))

((and (equal tool-1 tool-2) (equal tool-3 tool-4))
 (if (< left front)
     (list (list (list A D G H N L) tool-1 'L-enclosed-pocket)
        (list (list L M P O H J) tool-3 'L-slot))
   (list (list (list B D F O N M) tool-1 'L-slot)
      (list (list A B P F G J) tool-3 'L-enclosed-pocket))))

((and (equal tool-2 tool-3) (equal tool-4 tool-1))
 (if (< front right)
     (list (list (list C D G J K O) tool-2 'L-enclosed-pocket)
        (list (list A C N M P K) tool-4 'L-slot))
   (list (list (list N E G I P O) tool-2 'L-slot)
      (list (list A D E M I J) tool-4 'L-pocket))))

((equal tool-1 tool-2)
 (if (and (< left front) (< back front))
     (list (list (list A D G H N L) tool-1 'L-enclosed-pocket)
        pocket-3
        pocket-4)
   (if (< left front)
    (list (list (list A D F O N L) tool-1 'L-open-pocket)
          pocket-3
          pocket-4)
     (if (< back front)
      (list (list (list B D G H N M) tool-1 'L-open-pocket)
         pocket-3
         pocket-4)
     (list (list (list B D F O N M) tool-1 'L-slot)
           pocket-3
           pocket-4)))))

((equal tool-2 tool-3)
 (if (and (< left back) (< front back))
     (list (list (list C D G J K O) tool-2 'L-enclosed-pocket)
        pocket-4
        pocket-1)
   (if (< left back)
    (list (list (list N E G J K O) tool-2 'L-open-pocket)
          pocket-4
          pocket-1)
     (if (< front back)
      (list (list (list C D G I P O) tool-2 'L-open-pocket)
         pocket-4
         pocket-1)
     (list (list (list N E G I P O) tool-2 'L-slot)
```

```
                pocket-4
                pocket-1)))))

        ((equal tool-3 tool-4)
         (if (and (< right back) (< front back))
             (list (list (list A B P F G J) tool-3 'L-enclosed-pocket)
                pocket-1
                pocket-2)
           (if (< front back)
            (list (list (list A B P O H J) tool-3 'L-open-pocket)
                  pocket-1
                  pocket-2)
             (if (< right back)
              (list (list (list L M P F G J) tool-3 'L-open-pocket)
                 pocket-1
                 pocket-2)
             (list (list (list L M P O H J) tool-3 'L-slot)
                  pocket-1
                  pocket-2)))))

        ((equal tool-4 tool-1)
         (if (and (< right front) (< back front))
             (list (list (list A D E M I J) tool-4 'L-enclosed-pocket)
                pocket-2
                pocket-3)
           (if (< right front)
            (list (list (list A D E M P K) tool-4 'L-open-pocket)
                  pocket-2
                  pocket-3)
             (if (< back front)
              (list (list (list A C N M I J) tool-4 'L-open-pocket)
                 pocket-2
                 pocket-3)
             (list (list (list A C N M P K) tool-4 'L-slot)
                  pocket-2
                  pocket-3)))))


        (t (list pocket-1 pocket-2 pocket-3 pocket-4)))))



;;; This function is responsible for creating the machining
;;; regions for a layer with multiple islands.
;;;
;;; input:  metcapp-features  The list of MetCAPP features for
;;;                           that layer.
;;;         tools             The list of tool for that layer.
;;;
;;; output: A list of the machining regions for that layer.
;;;
(defun get-mult-machining-regions (metcapp-features tools)
```

```
  (let* ((blocks (car (last (car metcapp-features))))
      (same-layer-regions (add-tools metcapp-features
                          tools
                          nil))
      (same-layer-regions (add-identifier
                     (remove-intersections
                      0 0 same-layer-regions nil)
                     nil
                     0))
      (sorted-same-layer-regions (sort-by-tool-diameter
                          (find-adjacents
                           same-layer-regions
                           same-layer-regions
                           nil)
                          nil
                          nil))
      (same-tool-regions (get-same-tool-regions
                    sorted-same-layer-regions
                    nil))
      (sequenced-regions (sequence-regions
                    same-tool-regions
                    nil nil))
      (processed-regions (add-vertices-to-regions
                    sequenced-regions
                    blocks
                    nil))
      ) ; end of variable definitions
    (add-entry-sequences processed-regions 0 nil)
    )) ; end of function


;;; add-tools     recursive
;;; input:  same-layer-regions
;;;         tool-list
;;; output: list3 (same-layer-regions with metcapp-props replaced
;;;                by an appropriate tool-ID)
;;;
;;;
(defun add-tools (same-layer-regions tool-list list3)
  (cond ((null same-layer-regions) list3)

     (t (add-tools (cdr same-layer-regions)
                (cdr tool-list)
                (cons (list (car tool-list)
                      (second
                       (car same-layer-regions)))
                   list3))
        ) ; end of condition
     ) ; end of conditional
  ) ; end of function


;;; This function removes the intersection of two machining
```

```
;;; regions.
;;;
(defun remove-intersections (i j list1 list2)
  (let ((region1 (nth i list1))
     (region2 (nth j list1))
     )

    (cond ((null region1) (reverse list2))

       ((null region2)
        (remove-intersections (+ i 1)
                     0
                     list1
                     (cons region1
                           list2)))

       ((= i j)
        (remove-intersections i
                     (+ j 1)
                     list1
                     list2))

       ((> (car (get-mill-data (first region1)))
           (car (get-mill-data (first region2))))
        (remove-intersections i
                     (+ j 1)
                     list1
                     list2))

       ((null (intersection-bob (second region1)
                        (second region2)
                        nil))
        (remove-intersections i
                     (+ j 1)
                     list1
                     list2))

       (t (remove-intersections
           i
           0
           (append (bob-remove (list region1) list1)
                (reform-region (intersection-bob
                           (second region1)
                           (second region2)
                           nil)
                          (second region1)
                          (first region1)))
           list2))

       ) ; end of conditional
    ) ; end of let statement
  ) ; end of function
```

```
;;; This function is a child function of remove-intersections.
;;; intersection of region1 and region2
;;; list1 is the index list of region1
;;; list2 is the list of reformed reions
;;;
(defun reform-region (intersection list1 before1)
  (let* ((i-row-list    (remove-duplicates
                (get-all-nth 0 intersection nil)))
     (i-column-list (remove-duplicates
                (get-all-nth 1 intersection nil)))
     (row-list      (multiple-remove
                i-row-list
                (remove-duplicates
                 (get-all-nth 0 list1 nil))))
     (column-list   (multiple-remove
                i-column-list
                (remove-duplicates
                 (get-all-nth 1 list1 nil))))
     ) ; end of variable definitions
    (cond ((and (null row-list) (null column-list))
        nil)

       ((null row-list)
        (list (list before1
                (make-indexes i-row-list
                        column-list
                        nil))))

       ((null column-list)
        (list (list before1
                (make-indexes row-list
                        i-column-list
                        nil))))

       (t (list (list before1
                (make-indexes i-row-list
                          column-list
                          nil))
             (list before1
                (make-indexes row-list
                          i-column-list
                          nil))
             (list before1
                (make-indexes row-list
                          column-list
                          nil))
             ))
       ) ; end of conditional
    ) ; end of let statement
  ) ; end of function


;;; add-identifier     recursive
```

```
;;; child of get-mult-machining-regions function
;;;
;;; input:  sorted-same-layer-regions
;;; output: sorted-same-layer-regions where each region has a
;;;          unique integer identifier beginning with 0. This
;;;          will allow each region to be referenced by a
;;;          (nth n sorted-same-layer-regions) call.
;;;
(defun add-identifier (list1 list2 n)
  (cond ((null list1) (reverse list2))

     (t (add-identifier (cdr list1)
                  (cons (append (list n) (car list1))
                     list2)
                  (+ n 1)))
     ) ; end of conditional
  ) ; end of function


;;; list1 is a same-layer-region list such that each element has
;;;      the following structure:
;;;      (index tool-ID (block-list))
;;;      This list is processed by recursion.
;;;
;;; list2 is a same-layer-region list such that each element has
;;;      the following structure:
;;;      (index tool-ID (block-list))
;;;      This list is initially equal to list1 but is unaltered by
;;;      recursion.
;;;
;;; list3 is the new same-layer-region list such that each element
;;;      has the following structure:
;;;      (index tool-ID (block-list) (adjacent-regions-list)
;;;      (violated-sides-list))
;;;
(defun find-adjacents (list1 list2 list3)
  (let* ((row-list (sort (remove-duplicates
                   (get-all-nth 0
                            (third (car list1))
                            nil))
                '<))
      (column-list (sort (remove-duplicates
                    (get-all-nth 1
                            (third (car list1))
                            nil))
                   '<))

      (back (list 'back
                (make-indexes
                 (list (- (car row-list) 1))
                 column-list
                 nil)))
```

```
      (left (list 'left
                  (make-indexes
                   row-list
                   (list (- (car column-list) 1))
                   nil)))

      (front (list 'front
                   (make-indexes
                    (list (+ (car (last row-list)) 1))
                    column-list
                    nil)))

      (right (list 'right
                   (make-indexes
                    row-list
                    (list (+ (car (last column-list)) 1))
                    nil)))

      ) ; end of variable definitions

    (cond ((null (cdr list1))
           (reverse (cons (sub-find-adjacents
                           (car list1)
                           list2
                           (list left back right front))
                          list3)))

          (t (find-adjacents (cdr list1)
                             list2
                             (cons (sub-find-adjacents
                                    (car list1)
                                    list2
                                    (list left back right front))
                                   list3)))
          ) ; end of conditional
      ) ; end of let statement
  ) ; end of function


;;; sub-find-adjacents   Recursive child function of
;;;                      find-adjacents.
;;;
;;; region is an element of list1 as defined in the parent.
;;;
;;; list2 is list2 from the parent function.
;;;
;;; lbrf-list is the list of left back right front as defined in
;;;     the parent function for the region that is passed.
;;;
;;; new-region this is the result of processing region
;;; (the adjacents-list is added on).
;;; This will become an element of list3 in the parent function.
;;;
```

```
(defun sub-find-adjacents (region list2 lbrf-list)
  (let ((adjacentcy (get-adjacentcy lbrf-list
                          (third (car list2))))
     ) ; end of variable definitions

    (cond ((null list2) region)

       ((equal (car adjacentcy) 'fully)
        (sub-find-adjacents
         (list (first region)
            (second region)
            (third region)
            (cons (list (first (car list2))
                        (second adjacentcy)
                        (first region))
               (fourth region))
            (cons (second adjacentcy)
               (fifth region)))
         (cdr list2)
         lbrf-list)
        ) ; end of condition

       ((equal (car adjacentcy) 'partially)
        (sub-find-adjacents
         (list (first region)
            (second region)
            (third region)
            (cons (list (first (car list2))
                        (second adjacentcy)
                        (first region))
               (fourth region))
            (fifth region))
         (cdr list2)
         lbrf-list)
        ) ; end of condition

       (t (sub-find-adjacents region
                     (cdr list2)
                     lbrf-list))

       ) ; end of conditional
    ) ; end of let statement
  ) ; end of function


;;; list1 is the left-back-right-front list of the region for
;;; which adjacentcy is to be determined.
;;;
;;; list2 is the list of indecies of the region that adjacentcy-to
;;; is to be determined.
;;;
(defun get-adjacentcy (list1 list2)
  (cond ((null list1)
```

```
      (list 'none))

     ((bob-subsetp (second (car list1)) list2)
      (list 'fully (first (car list1))))

     ((not (null (intersection-bob (second (car list1))
                                   list2
                                   nil)))
      (list 'partially (first (car list1))))

     (t (get-adjacentcy (cdr list1) list2))
     ) ; end of conditional
  ) ; end of function


;;; sort-by-tool-diameter   recursive
;;; input:  same-layer-regions with tool-ID
;;;          ((identifier  tool-ID  (index-list)  (adjacents-list)
;;;           (violated-sides)) ...)
;;;
;;; output: same-layer-regions sorted such that the regions to be
;;;         cut by the largest tool diameters come first
;;;         (descending order).
;;;
(defun sort-by-tool-diameter (same-layer-regions
                              sorted-same-layer-regions
                              temp-list)
  (cond ((null same-layer-regions) sorted-same-layer-regions)

     ((and (null sorted-same-layer-regions)
           (null temp-list))
      (sort-by-tool-diameter (cdr same-layer-regions)
                             (list (car same-layer-regions))
                             nil))

     ((null sorted-same-layer-regions)
      (sort-by-tool-diameter
       (cdr same-layer-regions)
       (append (reverse temp-list)
               (list (car same-layer-regions)))
       nil))

     ((>= (car (get-mill-data
                (second (car same-layer-regions))))
          (car (get-mill-data
                (second (car sorted-same-layer-regions)))))
      (sort-by-tool-diameter
       (cdr same-layer-regions)
       (append (reverse temp-list)
               (cons (car same-layer-regions)
                     sorted-same-layer-regions))
       nil))
```

```
      (t (sort-by-tool-diameter
          same-layer-regions
          (cdr sorted-same-layer-regions)
          (cons (car sorted-same-layer-regions)
             (reverse temp-list))))
      ) ; end of conditional
  ) ; end of function


;;; get-same-tool-regions     recursive
;;; child of get-mult-machining-regions function
;;;
;;; input:  sorted-same-layer-regions
;;; output: list of same-tool-regions
;;;
(defun get-same-tool-regions (list1 list2)
  (cond ((null list1) (reverse list2))

      ((null list2)
       (get-same-tool-regions (cdr list1)
                    (list (list (car list1)))))

      ((equal (second (car list1))
           (second (car (car list2))))
       (get-same-tool-regions (cdr list1)
                    (cons (cons (car list1)
                                (car list2))
                          (cdr list2)))
       ) ; end of condition

      (t (get-same-tool-regions (cdr list1)
                       (cons (list (car list1))
                          list2)))
      ) ; end of conditional
  ) ; end of function


;;; list1 is the same-layer-regions list grouped by same tool
;;; next-list and list2 are initially nil
;;;
(defun sequence-regions (list1 next-list list2)
  (cond ((null list1) (reverse list2))

      (t (sequence-regions
          (cdr list1)
          (car (sub-sequence-regions (car list1)
                           next-list
                           nil
                           list2))
          (second (sub-sequence-regions (car list1)
                           next-list
                           nil
                           list2))))
```

```
    ) ; end of conditional
  ) ; end of function


(defun sub-sequence-regions (same-tool-regions
                             next-list1
                             next-list2
                             sequenced-regions)
  (cond ((null same-tool-regions)
         (list (append next-list1 next-list2)
               sequenced-regions))

        ((null next-list1)
         (sub-sequence-regions
          (cdr same-tool-regions)
          (fourth (car same-tool-regions))
          next-list2
          (cons (append (car same-tool-regions) '((enclosed)))
                sequenced-regions)))

        ((not (null (get-id-region (first (car next-list1))
                                   same-tool-regions)))
         (sub-sequence-regions
          (bob-remove (list (get-id-region
                             (first (car next-list1))
                             same-tool-regions))
                      same-tool-regions)
          (append (fourth (get-id-region (first (car next-list1))
                                         same-tool-regions))
                  (bob-remove (list (car next-list1)) next-list1))
          next-list2
          (cons (append (get-id-region (first (car next-list1))
                                       same-tool-regions)
                        (list (car next-list1)))
                sequenced-regions)))

        (t (sub-sequence-regions
            same-tool-regions
            (bob-remove (list (car next-list1)) next-list1)
            (cons (car next-list1) next-list2)
            sequenced-regions))
        ) ; end of conditional
  ) ; end of function


(defun add-vertices-to-regions (regions blocks list2)
  (cond ((null regions) (reverse list2))
        (t (add-vertices-to-regions
            (cdr regions)
            blocks
            (cons (list (first (car regions))
                        (second (car regions))
                        (get-region-vertices (third (car regions))
```

```
                                blocks)
                (fourth (car regions))
                (fifth (car regions))
                (sixth (car regions)))
              list2)))
    )
  )


(defun get-region-vertices (indices blocks)
  (let* ((rows (sort (remove-duplicates
                 (get-all-nth 0 indices nil)) '<))
      (columns (sort (remove-duplicates
                 (get-all-nth 1 indices nil)) '<))
    )
    (list (apt-from-xz (second (nth (first columns)
                         (nth (car (last rows))
                           blocks))))
       (apt-from-xz (fourth (nth (car (last columns))
                         (nth (car (last rows))
                           blocks))))
       (apt-from-xz (third  (nth (car (last columns))
                         (nth (first rows)
                           blocks))))
       (apt-from-xz (first  (nth (first columns)
                         (nth (first rows)
                           blocks))))
       )
    )
  )


;;; i is a counter set initially to 0
;;; region-2 is the region being entered
;;;
(defun add-entry-sequences (processed-regions i list2)
  (let* ((region-2 (nth i processed-regions))
      (region-1 (get-id-region (third (sixth region-2))
                       processed-regions))
    ) ; end of variable definitions

    (cond ((null region-2) (reverse list2))

       ((null region-1)
        (add-entry-sequences processed-regions
                       (+ i 1)
                       (cons (list (second region-2)
                                   (third  region-2)
                                   'enclosed)
                          list2)))

       (t (add-entry-sequences
           processed-regions
```

```
            (+ i 1)
            (cons (list (second region-2)
                  (third  region-2)
                  (get-entry-sequence
                   (sixth  region-2)
                   (third  region-1)
                   (third  region-2)
                   (dietrick-round
                    (/ (car (get-mill-data
                          (second region-2)))
                       25.4))))
                  list2)))
        ) ; end of conditional
    ) ; end of let statement
  ) ; end of function


;;; entry is the list of "how the region is to be entered"
;;;    (region-being-entered  opposite-side-being-entered
;;;     region-being-exited)
;;;
;;; vertices-1 is the list of vertices of the region being exited
;;; vertices-2 is the list of vertices of the region being entered
;;;
(defun get-entry-sequence (entry vertices-1 vertices-2 tool-dia)
  (cond ((equal (second entry) 'front)
      (get-entry-into-back vertices-1 vertices-2 tool-dia))

     ((equal (second entry) 'back)
      (get-entry-into-front vertices-1 vertices-2 tool-dia))

     ((equal (second entry) 'left)
      (get-entry-into-right vertices-1 vertices-2 tool-dia))

     ((equal (second entry) 'right)
      (get-entry-into-left vertices-1 vertices-2 tool-dia))

     ) ; end of conditional
  ) ; end of function


;;; vertices-1 refers to the vertices of the pocket-entered-from
;;; vertices-2 refers to the vertices of the pocket-entered
;;;
;;; alphas refer to pocket being exited
;;; betas refer to pocket being entered
;;;
(defun get-entry-into-back (vertices-1 vertices-2 tool-dia)
  (let* ((alpha-1 (first  vertices-1))
      (alpha-2 (second vertices-1))
      (beta-1  (fourth vertices-2))
      (beta-2  (third  vertices-2))
      (delta   (cond ((and (<= (first beta-1) (first alpha-1))
```

```
                    (<= (first alpha-2) (first beta-2)))
                (list (/ (+ (first alpha-1)
                            (first alpha-2))
                         2.0)
                         (second alpha-1)))

               ((and (<= (first alpha-1) (first beta-1))
                     (<= (first beta-2) (first alpha-2)))
                (list (/ (+ (first beta-1)
                            (first beta-2))
                         2.0)
                         (second beta-1)))

               ((and (<= (first alpha-1) (first beta-1))
                     (<= (first beta-1) (first alpha-2)))
                (list (/ (+ (first beta-1)
                            (first alpha-2))
                         2.0)
                         (second beta-1)))

               ((and (<= (first alpha-1) (first beta-2))
                     (<= (first beta-2) (first alpha-2)))
                (list (/ (+ (first beta-2)
                            (first alpha-1))
                         2.0)
                         (second alpha-1)))
               ) ; end of condional
            ) ; end of delta definition

      (gamma (list (dietrick-round
                    (/ (+ (first (first vertices-2))
                          (first (second vertices-2))) 2.0))
                   (dietrick-round
                    (/ (+ (second (first vertices-2))
                          (second (third vertices-2))) 2.0))))

      ) ; end of variable definitions

    (list (list (first delta)
                (+ (second delta) (* 0.5 tool-dia)))
          (list (first delta)
                (- (second delta) (* 0.5 tool-dia)))
          gamma)
    ) ; end of let statement
  ) ; end of function


;;; vertices-1 refers to the vertices of the pocket-entered-from
;;; vertices-2 refers to the vertices of the pocket-entered
;;;
(defun get-entry-into-front (vertices-1 vertices-2 tool-dia)
  (let* ((alpha-1 (fourth vertices-1))
         (alpha-2 (third  vertices-1))
```

```
      (beta-1  (first  vertices-2))
      (beta-2  (second vertices-2))
      (delta   (cond ((and (<= (first beta-1) (first alpha-1))
                           (<= (first alpha-2) (first beta-2)))
                      (list (/ (+ (first alpha-1)
                                  (first alpha-2))
                               2.0)
                               (second alpha-1)))

                     ((and (<= (first alpha-1) (first beta-1))
                           (<= (first beta-2) (first alpha-2)))
                      (list (/ (+ (first beta-1)
                                  (first beta-2))
                               2.0)
                               (second beta-1)))

                     ((and (<= (first alpha-1) (first beta-1))
                           (<= (first beta-1) (first alpha-2)))
                      (list (/ (+ (first beta-1)
                                  (first alpha-2))
                               2.0)
                               (second beta-1)))

                     ((and (<= (first alpha-1) (first beta-2))
                           (<= (first beta-2) (first alpha-2)))
                      (list (/ (+ (first beta-2)
                                  (first alpha-1))
                               2.0)
                               (second alpha-1)))
                     ) ; end of condional
            ) ; end of delta definition

      (gamma (list (dietrick-round
                    (/ (+ (first (first vertices-2))
                          (first (second vertices-2))) 2.0))
                   (dietrick-round
                    (/ (+ (second (first vertices-2))
                          (second (third vertices-2))) 2.0))))

      ) ; end of variable definitions

    (list (list (first delta)
                (- (second delta) (* 0.5 tool-dia)))
          (list (first delta)
                (+ (second delta) (* 0.5 tool-dia)))
          gamma)
    ) ; end of let statement
  ) ; end of function


;;; entry refers to the list "how will this pocket be entered"
;;;  (pocket-entered  opposite-side-entered  pocket-entered-from)
;;; vertices-1 refers to the vertices of the pocket-entered-from
```

```
;;; vertices-2 refers to the vertices of the pocket-entered
;;;
(defun get-entry-into-left (vertices-1 vertices-2 tool-dia)
  (let* ((alpha-1 (second vertices-1))
      (alpha-2 (third  vertices-1))
      (beta-1  (first  vertices-2))
      (beta-2  (fourth vertices-2))
      (delta   (cond ((and (<= (second beta-1)
                           (second alpha-1))
                           (<= (second alpha-2)
                           (second beta-2)))
                 (list (first alpha-1)
                       (/ (+ (second alpha-1)
                             (second alpha-2))
                       2.0)))

                ((and (<= (second alpha-1)
                           (second beta-1))
                           (<= (second beta-2)
                           (second alpha-2)))
                 (list (first beta-1)
                       (/ (+ (second beta-1)
                             (second beta-2))
                       2.0)))

                ((and (<= (second alpha-1)
                           (second beta-1))
                           (<= (second beta-1)
                           (second alpha-2)))
                 (list (first beta-1)
                       (/ (+ (second beta-1)
                             (second alpha-2))
                       2.0)))

                ((and (<= (second alpha-1)
                           (second beta-2))
                           (<= (second beta-2)
                           (second alpha-2)))
                 (list (first alpha-1)
                       (/ (+ (second beta-2)
                             (second alpha-1))
                       2.0)))
               ) ; end of condional
            ) ; end of delta definition

      (gamma (list (dietrick-round
                    (/ (+ (first (first vertices-2))
                          (first (second vertices-2)))
                     2.0))
                   (dietrick-round
                    (/ (+ (second (first vertices-2))
                          (second (third vertices-2)))
                     2.0))))
```

```
    ) ; end of variable definitions

   (list (list (- (first delta) (* 0.5 tool-dia))
         (second delta))
      (list (+ (first delta) (* 0.5 tool-dia))
         (second delta))
      gamma)
   ) ; end of let statement
  ) ; end of function


;;; vertices-1 refers to the vertices of the pocket-entered-from
;;; vertices-2 refers to the vertices of the pocket-entered
;;;
(defun get-entry-into-right (vertices-1 vertices-2 tool-dia)
  (let* ((alpha-1 (first  vertices-1))
      (alpha-2 (fourth vertices-1))
      (beta-1  (second vertices-2))
      (beta-2  (third  vertices-2))
      (delta   (cond ((and (<= (second beta-1)
                           (second alpha-1))
                         (<= (second alpha-2)
                           (second beta-2)))
                  (list (first alpha-1)
                         (/ (+ (second alpha-1)
                            (second alpha-2))
                           2.0)))

                 ((and (<= (second alpha-1)
                           (second beta-1))
                         (<= (second beta-2)
                           (second alpha-2)))
                  (list (first beta-1)
                         (/ (+ (second beta-1)
                            (second beta-2))
                           2.0)))

                 ((and (<= (second alpha-1)
                           (second beta-1))
                         (<= (second beta-1)
                           (second alpha-2)))
                  (list (first beta-1)
                         (/ (+ (second beta-1)
                            (second alpha-2))
                           2.0)))

                 ((and (<= (second alpha-1)
                           (second beta-2))
                         (<= (second beta-2)
                           (second alpha-2)))
                  (list (first alpha-1)
                         (/ (+ (second beta-2)
                            (second alpha-1))
```

```
                    2.0)))
               ) ; end of condional
          ) ; end of delta definition

    (gamma (list (dietrick-round
                (/ (+ (first (first vertices-2))
                      (first (second vertices-2))) 2.0))
               (dietrick-round
                (/ (+ (second (first vertices-2))
                      (second (third vertices-2))) 2.0))))

    ) ; end of variable definitions

  (list (list (+ (first delta) (* 0.5 tool-dia))
        (second delta))
     (list (- (first delta) (* 0.5 tool-dia))
        (second delta))
     gamma)
  ) ; end of let statement
) ; end of function
```