6
Reverse Engineering and Inspection of Machined Parts in Manufacturing Systems

6.1 Introduction

6.2 Objectives and Questions

6.3 Sensing for Inspection and Reverse Engineering
Discrete Event Dynamic Systems • Sensing Strategy • The Dynamic Recursive Context for Finite State Machines

6.4 Sensory Processing
Two-dimensional Image Processing • Visual Observation of States • Deciding Feature Relationships • Constructing the Recursive Relation • Extraction of Depth Information and World Coordinates • Camera Calibration • Depth Estimation Using an Illumination Table

6.5 Sensing to CAD Interface
Contours to Splines • Contours to Machined Features

6.6 System Integration
The Initiation Experiment • The Second Experiment • Running the Experiment • Experimental Results, Automated Inspection

6.7 Summary of Current Developments
Goals and Methodology • Current Developments • Past, Current, and Future Activities

6.8 Integration Efforts
Robotics and Sensing • Computer Aided Design and Manufacturing • VLSI, Uncertainty Modeling, and Languages

6.9 Conclusions

Appendix A

Appendix B

Appendix C

Tarek Sobh
University of Bridgeport

Jonathan Owen
University of Bridgeport

Mohamed Dekhil
Rain Infinity

We discuss a design for inspection and reverse engineering environments. We investigate the use of the dynamic recursive context of discrete event dynamic systems (DRFSM DEDS) to guide and control the active exploration and sensing of mechanical parts for industrial inspection and reverse engineering, and utilize the recursive nature of the parts under consideration. In our work, we construct a sensing to CAD interface for the automatic reconstruction of parts from visual data. This chapter includes results and describes this interface in detail, demonstrating its effectiveness with reverse-engineered machined parts.

6.1 Introduction

Developing frameworks for inspection and reverse-engineering applications is an essential activity in many engineering disciplines. Usually, too much time is spent in designing hardware and software environments in order to be able to attack a specific problem. One of the purposes of this work is to provide a basis for solving a class of inspection and reverse-engineering problems.

CAD/CAM (computer aided design/manufacturing) typically involves the design and manufacture of mechanical parts. The problem of reverse engineering is to take an existing mechanical part as the point of departure and to inspect or produce a design, and perhaps a manufacturing process, for the part. The techniques that we explore can be used for a variety of applications. We use an observer agent to sense the current world environment and make some measurements, then supply relevant information to a control module that will be able to make some design choices that will later affect manufacturing and/or inspection activities. This involves both autonomous and semi-autonomous sensing.

The problem of inspection typically involves using a CAD representation of the item to be inspected, and using it to drive an inspection tool such as the coordinate measuring machine (CMM). An example of this is found in [9]. While the work described there is intended to allow the inspection of complicated sculpted surfaces, we limit ours to an important class of machined parts. Within this class, we hope to reduce the time necessary for inspection by more than tenfold, taking advantage of the part's recursive nature and its feature description.

We use a recursive dynamic strategy for exploring machine parts. A discrete event dynamic system (DEDS) framework is designed for modeling and structuring the sensing and control problems. The dynamic recursive context for finite state machines (DRFSM) is introduced as a new DEDS tool for utilizing the recursive nature of the mechanical parts under consideration. This chapter describes what this means in more detail.

6.2 Objectives and Questions

The objective of this research project is to explore the basis for a consistent software and hardware environment, and a flexible system that is capable of performing a variety of inspection and reverse-engineering activities. In particular, we will concentrate on the adaptive automatic extraction of some properties of the world to be sensed and on the subsequent use of the sensed data for producing reliable descriptions of the sensed environments for manufacturing and/or description refinement purposes. We use an observer agent with some sensing capabilities (vision and touch) to actively gather data (measurements) of mechanical parts. We conjecture that discrete event dynamical systems (DEDS) provide a good base for defining consistent and adaptive control structures for the sensing module of the inspection and reverse-engineering problem. If this is true, then we will be able to answer the following questions:

- What is a suitable algorithm to coordinate sensing, inspection, design, and manufacturing?
- What is a suitable control strategy for sensing the mechanical part?
- Which parts should be implemented in hardware vs. software?
- What are suitable language tools for constructing a reverse-engineering and/or inspection strategy?

DEDS can be simply described as: dynamic systems (typically asynchronous) in which state transitions are triggered by discrete events in the system.

It is possible to *control* and *observe* hybrid systems (systems that involve continuous, discrete, and symbolic parameters) under uncertainty using DEDS formulations [13, 16].

The applications of this work are numerous: automatic inspection of mechanical or electronic components and reproduction of mechanical parts. Moreover, the experience gained in performing this research will allow us to study the subdivision of the solution into reliable, reversible, and easy-to-modify software and hardware environments.

6.3 Sensing for Inspection and Reverse Engineering

This section describes the solution methodology for the sensing module and discusses the components separately. The control flow is described and the methods, specific equipment, and procedures are also discussed in detail.

We use a vision sensor (B/W CCD camera) and a coordinate measuring machine (CMM) with the necessary software interfaces to a Sun Sparcstation as the sensing devices. The object is to be inspected by the cooperation of the observer camera and the probing CMM. A DEDS is used as the high-level framework for exploring the mechanical part. A dynamic recursive context for finite state machines (DRFSM) is used to exploit the recursive nature of the parts under consideration.

Discrete Event Dynamic Systems

DEDS are usually modeled by finite state automata with partially observable events, together with a mechanism for enabling and disabling a subset of state transitions [3, 12, 13]. We propose that this model is a suitable framework for many reverse-engineering tasks. In particular, we use the model as a high-level structuring technique for our system.

We advocate an approach in which a stabilizable semi-autonomous visual sensing interface would be capable of making decisions about the *state* of the observed machine part and the probe, thus providing both symbolic and parametric descriptions to the reverse-engineering and/or inspection control module. The DEDS-based active sensing interface is discussed in the following section.

Modeling and Constructing an Observer

The tasks that the autonomous observer system executes can be modeled efficiently within a DEDS framework. We use the DEDS model as a high-level structuring technique to preserve and make use of the information we know about the way in which a mechanical part should be explored. The state and event description is associated with different visual cues; for example, appearance of objects, specific 3-D movements and structures, interaction between the touching probe and part, and occlusions. A DEDS observer serves as an intelligent sensing module that utilizes existing information about the tasks and the environment to make informed tracking and correction movements and autonomous decisions regarding the state of the system.

To know the current state of the exploration process, we need to observe the sequence of events occurring in the system and make decisions regarding the state of the automaton. State ambiguities are allowed to occur; however, they are required to be resolvable after a bounded interval of events. The goal will be to make the system a strongly output-stabilizable one and/or construct an observer to satisfy specific task-oriented visual requirements. Many 2-D visual cues for estimating 3-D world behavior can be used. Examples include image motion, shadows, color, and boundary information. The uncertainty in the sensor acquisition procedure and in the image processing mechanisms should be taken into consideration to compute the world uncertainty.

Foveal and peripheral vision strategies could be used for the autonomous "focusing" on relevant aspects of the scene. Pyramid vision approaches and logarithmic sensors could be used to reduce the dimensionality and computational complexity for the scene under consideration.

Error States and Sequences

We can utilize the observer framework for recognizing error states and sequences. The idea behind this recognition task is to be able to report on *visually incorrect* sequences. In particular, if there is a predetermined observer model of a particular inspection task under observation, then it would be useful to determine if something goes wrong with the exploration actions. The goal of this reporting procedure is to alert the operator or autonomously supply feedback to the inspecting robot so that it can correct its actions. An example of errors in inspection is unexpected occlusions between the observer camera and the inspection environment, or probing the part in a manner that might break the probe. The correct sequences of automata state transitions can be formulated as the set of strings that are *acceptable* by the observer automaton. This set of strings represents precisely the language describing all possible visual task evolution steps.

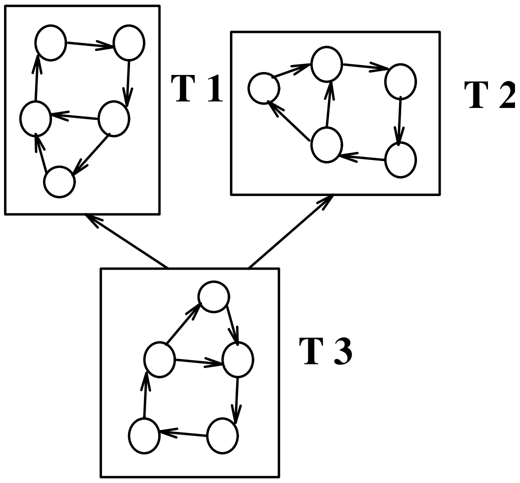


FIGURE 6.1 Hierarchy of tasks.

Hierarchical Representation

Figure 6.1 shows a hierarchy of three submodels. Motives behind establishing hierarchies in the DEDS modeling of different exploration tasks include reducing the search space of the observer and exhibiting modularity in the controller design. This is done through the designer, who subdivides the task space of the exploring robot into separate submodels that are inherently independent. Key events cause the transfer of the observer control to new submodels within the hierarchical description. Transfer of control through the observer hierarchy of models allows coarse to fine shift of attention in recovering events and asserting state transitions.

Mapping Module

The object of having a mapping module is to dispense with the need for the manual design of DEDS automata for various platform tasks. In particular, we would like to have an off-line module, which is to be supplied with some symbolic description of the task under observation and whose output would be the code for a DEDS automaton that is to be executed as the observer agent. A graphical representation of the mapping module is shown in Fig. 6.2. The problem reduces to figuring out what is an appropriate form for the task description. The error state paradigm motivated regarding this problem as the inverse problem of determining acceptable languages for a specific DEDS observer automaton. In particular, we suggest a skeleton for the mapping module that transforms a collection of input strings into an automaton model.

The idea is to supply the mapping module with a collection of strings that represents possible state transition sequences. The input highly depends on the task under observation, what is considered as relevant states and how coarse the automaton should be. The sequences are input by an operator. It should be obvious that the "garbage-in-garbage-out" principle holds for the construction process; in particular, if the set of input strings is not representative of all possible scene evolutions, then the automaton would be a faulty one. The experience and knowledge that the operator have would influence

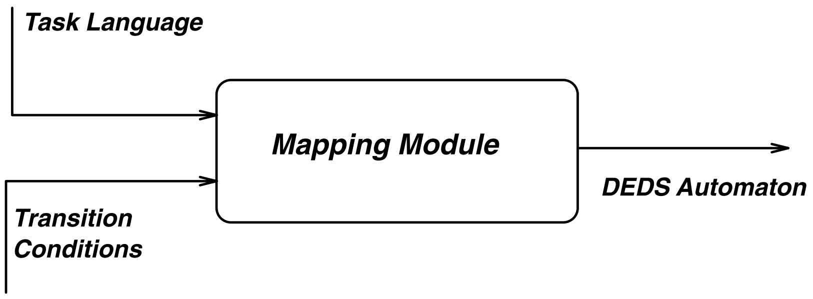


FIGURE 6.2 The mapping module.

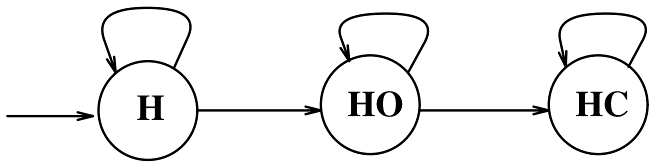


FIGURE 6.3 An automaton for simple grasping.

the outcome of the resulting model. However, it should be noticed that the level of experience needed for providing these sets of strings is much lower than the level of experience needed for a designer to actually construct a DEDS automaton manually. The description of the events that cause transitions between different symbols in the set of strings should be supplied to the module in the form of a list.

As an illustrative example, suppose that the task under consideration is simple grasping of one object and that all we care to know is three configurations: whether the hand is alone in the scene, whether there is an object in addition to the hand, and whether enclosure has occurred. If we represent the configurations by three states h, h_o, and h_c, then the operator would have to supply the mapping module with a list of strings in a language, whose alphabet consists of those three symbols, and those strings should span the entire language, so that the resulting automaton would accept all possible configuration sequences. The mapping from a set of strings in a regular language into a minimal equivalent automaton is a solved problem in automata theory.

One possible language to describe this simple automaton is:

$$L = h^+ h_o^+ h_c^+$$

and a corresponding DEDS automaton is shown in Fig. 6.3.

The best-case scenario would have been for the operator to supply exactly the language L to the mapping module with the appropriate event definitions. However, it could be the case that the set of strings that the operator supplies does not represent the task language correctly, and in that case some learning techniques would have to be implemented which, in effect, augment the input set of strings into a language that satisfies some predetermined criteria. For example, y^* is substituted for any string of y's having a length greater than n, and so on. In that case, the resulting automaton would be correct up to a certain degree, depending on the operator's experience and the correctness of the learning strategy.

Sensing Strategy

We use a B/W CCD camera mounted on a robot arm and a coordinate measuring machine (CMM) to sense the mechanical part. A DRFSM implementation of a discrete event dynamic system (DEDS) algorithm is used to facilitate the state recovery of the inspection process. DEDS is suitable for modeling robotic observers as it provides a means for tracking the *continuous, discrete*, and *symbolic* aspects of the scene under consideration [3, 12, 13]. Thus, the DEDS controller will be able to *model* and *report* the state evolution of the inspection process.

In inspection, the DEDS guides the sensing machines to the parts of the objects where discrepancies occur between the real object (or a CAD model of it) and the recovered structure data points and/or parameters. The DEDS formulation also compensates for noise in the sensor readings (both ambiguities and uncertainties) using a probabilistic approach for computing the 3-D world parameters [16]. The recovered data from the sensing module is then used to drive the CAD module. The DEDS sensing agent is thus used to collect data of a *passive* element for designing *structures*; an exciting extension is to use a similar DEDS observer for moving agents and subsequently design *behaviors* through a learning stage.

The Dynamic Recursive Context for Finite State Machines

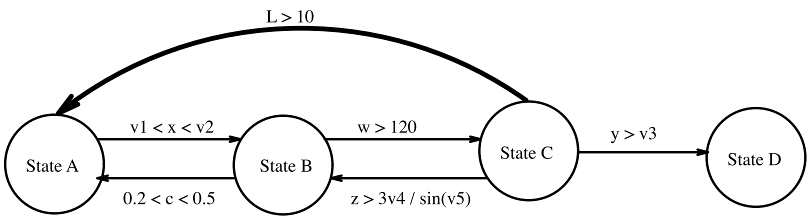
The dynamic recursive context for finite state machines (DRFSM) is a new methodology to represent and implement multi-level recursive processes using systematic implementation techniques. By multi-level process, we mean any processing operations that are done repetitively with different parameters. DRFSM has proved to be a very efficient way to solve many complicated problems in the inspection paradigm using an easy notation and a straightforward implementation, especially for objects that have similar multi-level structures with different parameters. The main idea of the DRFSM is to reuse the conventional DEDS finite state machine for a new level after changing some of the transition parameters. After exploring this level, it will retake its old parameters and continue exploring the previous levels. The implementation of such machines can be generated automatically by some modification to an existing reactive behavior design tool, called GIJoe [4], that is capable of producing code from state machine descriptions (drawings) by adding a recursive representation to the conventional representation of finite state machines, and then generating the appropriate code for it.

Definitions

Variable transition value: Any variable value that depends on the level of recursion.
Variable transition vector: The vector containing all variable transitions values, and is dynamically changed from level to level.
Recursive state: A state calling another state recursively, and this state is responsible for changing the variable transition vector to its new value according to the new level.
Dead-end state: A state that does not call any other state (no transition arrows come out of it). In DRFSM, when this state is reached, it means to go back to a previous level, or quit if it is the first level. This state is usually called the error-trapping state. It is desirable to have several dead-end states to represent different types of errors that can happen in the system.

DRFSM Representation

We will use the same notation and terms of the ordinary FSMs, but some new notation to represent recursive states and variable transitions. First, we permit a new type of transition, as shown in Fig. 6.4, (from state C to A); this is called the recursive transition (RT). A recursive transition arrow (RTA) from one state to another means that the transition from the first state to the second state is done by a recursive call to the second one after changing the variable transition vector. Second, the transition condition



trans. Variables	V1	V2	V3	V4	V5
Level 1	12	15	0.03	170	25
Level 2	10	12	0.07	100	35
Level 3	6	8	0.15	50	40

FIGURE 6.4 A simple DRFSM.

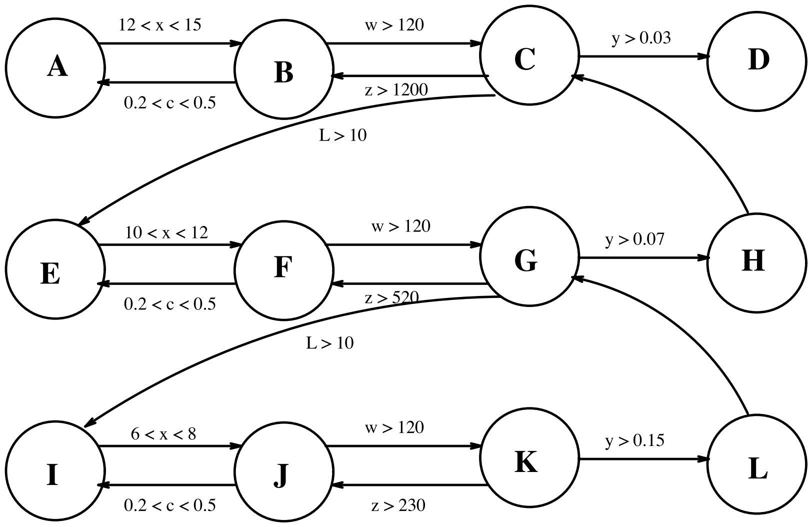


FIGURE 6.5 Flat representation of a simple DRFSM.

from a state to another may contain variable parameters according to the current level; these variable parameters are distinguished from the constant parameters by the notation V (parameter name). All variable parameters of all state transitions constitute the variable transition vector. It should be noticed that nondeterminism is not allowed, in the sense that it is impossible for two concurrent transitions to occur from the same state. Figure 6.5 is the equivalent FSM representation (or the flat representation) of the DRFSM shown in Fig. 6.4, for three levels, and it illustrates the compactness and efficiency of the new notation for this type of process.

A Graphical Interface for Developing DRFSMs

In developing the framework for reverse engineering, it has proven desirable to have a quick and easy means of modifying the DRFSM that drives the inspection process. This was accomplished by modifying an existing reactive behavior design tool, GIJoe, to accommodate producing the code of DRFSM DEDS.

GIJoe [4] allows the user to graphically draw finite state machines, and output the results as C code. GIJoe's original method was to parse transition strings using lex/yacc-generated code. The user interface is quite intuitive, allowing the user to place states with the left mouse button, and transitions by selecting the start and end states with left and right mouse buttons. When the state machine is complete, the user selects a state to be the start state and clicks the Compile button to output C code.

The code output by the original GIJoe has an iterative structure that is not conducive to the recursive formulation of dynamic recursive finite state machines. Therefore, it was decided to modify GIJoe to suit our needs. Modifications to GIJoe include:

- Output of recursive rather than iterative code to allow recursive state machines
- Modification of string parsing to accept recursive transition specification
- Encoding of an event parser to prioritize incoming events from multiple sources
- Implementation of the variable transition vector (VTV) acquisition (when making recursive transitions)

Example code from the machine in Fig. 6.6 can be found in Appendix A. We used this machine in our new experiment, which will be mentioned in a later section.

The event parser was encoded to ensure that the automaton makes transitions on only one source of input. Currently acceptable events are as follows:

- Probe: probe is in the scene
- NoProbe: no probe is in the scene

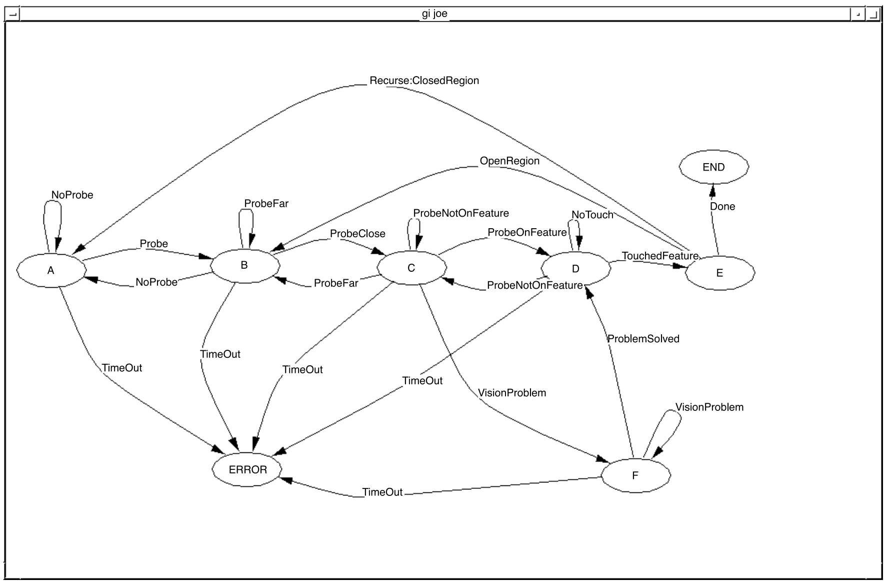


FIGURE 6.6 GIJoe window w/DRFSM.

- ProbeClose: probe is within the "close" tolerance to the current feature specified by the VTV
- ProbeFar: probe is farther from the current feature than the "close" tolerance specified by the VTV
- ProbeOnFeature: probe is on the feature (according to vision)
- ProbeNotOnFeature: probe is close, but not on the feature (according to vision.)
- VisionProblem: part string has changed, signifying that a feature is occluded (need to move the camera)
- ProblemSolved: moving the camera has corrected the occlusion problem
- TouchedFeature: probe has touched the feature (according to touch sensor)
- NoTouch: probe has not touched the feature (according to touch sensor)
- ClosedRegion: current feature contains closed region(s) to be inspected (recursively)
- OpenRegion: current feature contains open region(s) to be inspected (iteratively)
- TimeOut: machine has not changed state within a period of time specified by the VTV
- Done: inspection of the current feature and its children is complete; return to previous level

Additional events require the addition of suitable event handlers. New states and transitions may be added completely within the GIJoe interface. The new code is output from GIJoe and may be linked to the inspection utilities with no modifications.

The VTV, or variable transition vector, is a vector containing variables that may be dependent on the current depth of recursion. It is currently read from a file.

The code produced by the machine in Fig. 6.6 was first tested using a text interface before being linked with the rest of the experimental code. The following is a transcript showing the simulated exploration of two closed regions A and B, with A containing B:

```
inspect [5] ~/DEDS => bin/test_drfsm
   enter the string: A(B())
A(B())

               THE VARIABLE TRANSITION VECTOR

         100.000000          50.000000
in state A
```

```
   has the probe appeared? n                          % NoProbe is true
   has the probe appeared? n
   has the probe appeared? y                          % Probe is true
in state B
   has the probe appeared? y
   enter the distance from probe to A: 85             % Probe is Far
   has the probe appeared? y
   enter the distance from probe to A: 45             % Probe is Close
in state C
   enter the string: A (B())
   enter the distance from probe to A: 10
   is the probe on A? y
in state D
   is the probe on A? y                               % Probe on Feature
   has touch occurred? y
in state E
Making recursive call...


          THE VARIABLE TRANSITION VECTOR


      100.000000       50.000000

in state A
   has the probe appeared? y                          % Probe is true
in state B
   has the probe appeared? y
   enter the distance from probe to B: 95             % Probe is Far
   has the probe appeared? y
   enter the distance from probe to B: 45             % Probe is Close
in state C
   enter the string: A(B())
   enter the distance from probe to B: 10
   is the probe on B? y
in state D
   is the probe on B? y                               % Probe on Feature
   has touch occurred? y
in state E
in state END
in state END

Inspection Complete.

inspect[6] ~/DEDS =>
```

The obtained results, when linked with the rest of the experimental code, were as expected. Future modifications may include the addition of "output" on transitions, such as "TouchOccurred/Update-Model," allowing easy specification of communication between modules. It should be clear, however, that the code generated by GIJoe is only a skeleton for the machine, and has to be filled by the users according to the tasks assigned to each state.

In general, GIJoe proved to be a very efficient and handy tool for generating and modifying such machines. By automating code generation, one can reconfigure the whole inspection process without being familiar with the underlying code (given that all required user-defined events and modules are available).

How to Use DRFSM

To apply DRFSM to any problem, the following steps are required:

1. Problem analysis: divide the problem into states, so that each state accomplishes a simple task.
2. Transition conditions: find the transition conditions between the states.
3. Explore the repetitive part in the problem (recursive property) and specify the recursive states. However, some problems may not have this property; in those cases, a FSM is a better solution.
4. VTV formation: if there are different transition values for each level, these variables must be defined.
5. Error trapping: using robust analysis, a set of possible errors can be established; then, one or more dead-end state(s) are added.
6. DRFSM design: use GIJoe to draw the DRFSM and generate the corresponding C code.
7. Implementation: the code generated by GIJoe has to be filled out with the exact task of each state, the error handling routines should be written, and the required output must be implemented as well.

Applying DRFSM in Features Extraction

An experiment was performed for inspecting a mechanical part using a camera and a probe. A predefined DRFSM state machine was used as the observer agent skeleton. The camera was mounted on a PUMA 560 robot arm so that the part was always in view. The probe could then extend into the field of view and come into contact with the part, as shown in Fig. 6.19.

Symbolic representation of features: for this problem, we are concerned with open regions (O) and closed regions (C). Any closed region may contain other features (the recursive property). Using parenthesis notation, the syntax for representing features can be written as follows:

$$\begin{aligned}
&< \text{feature} > :: \text{C}(< \text{subfeature} >)\ |\text{C}()\\
&< \text{subfeature} > :: < \text{term} >, < \text{subfeature} >\ |< \text{term} >\\
&< \text{term} > :: \text{O}\ |< \text{feature} >
\end{aligned}$$

For example, the symbolic notation of Fig. 6.7 is:

C(O,C(O,C(),C(O)),C())

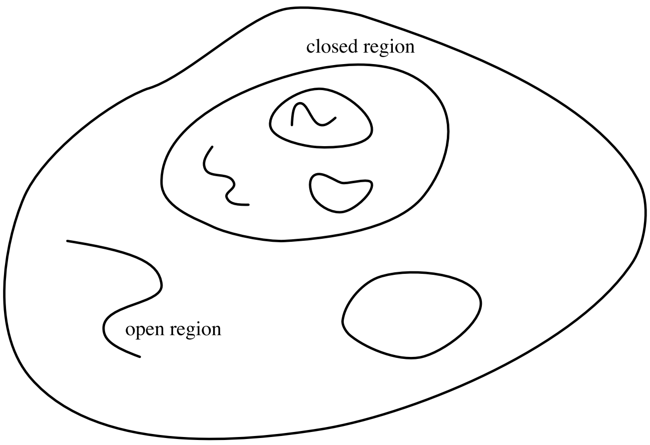


FIGURE 6.7 An example for a recursive object.

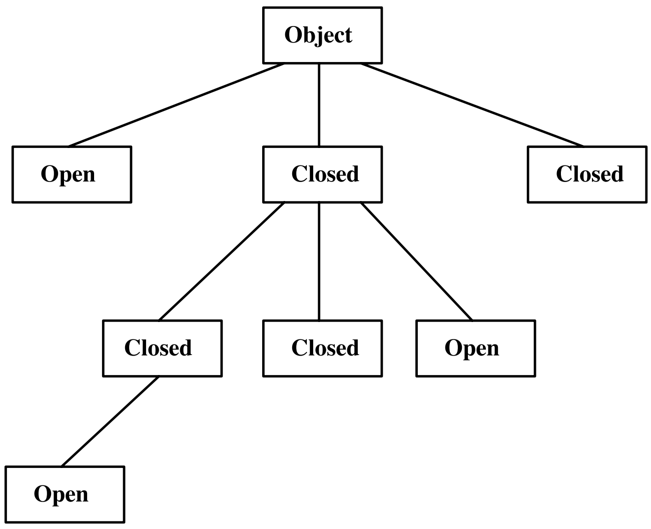


FIGURE 6.8 Graph for the recursive object.

Figure 6.8 shows the graphical representation of this recursive structure which is a tree-like structure. Future modifications to DRFSMs includes allowing different functions for each level.

6.4 Sensory Processing

For the state machine to work, it must be aware of state changes in the system. As inspection takes place, the camera supplies images that are interpreted by a vision processor and used to drive the DRFSM.

A B/W CCD camera is mounted on the end effector of a Puma 560 robot arm. The robot is then able to position the camera in the workplace, take stereo images, and move in the case of occlusion problems. The part to be inspected is placed on the coordinate measuring machine (CMM) table. The probe then explores the part while the mobile camera is able to observe the inspection and provide scene information to the state machine.

The vision system provides the machine with specific information about the current state of the inspection. This is done through several layers of vision processing and through the cooperation of 2-D, $2\frac{1}{2}$-D, and 3-D vision processors.

The aspects of the image that need to be given to the state machine include:

- Number of features
- Contour representation of each feature
- Relationships of the features
- Depth of features
- Approximate depth estimates between features
- Location of the probe with respect to the part

Two-dimensional Image Processing

Two-dimensional (2-D) features of the part are extracted using several different visual image filters. The camera captures the current image and passes it to the 2-D image processing layer. After the appropriate processing has taken place, important information about the scene is supplied to the other vision layers and the state machine.

The images are captured using a B/W CCD camera, which supplies 640 $\times$ 480 pixels to a VideoPix video card in a Sun Workstation. The 2-D processing is intended to supply a quick look at the current state of the inspection and only needs a single image, captured without movement of the Puma.

The images are copied from the video card buffer and then processed. The main goal of image processing modules is to discover features. Once features are discovered, contours are searched for among the feature responses.

Extracting Contours

Contours are considered important features that supply the machine with information necessary to build an object model and drive the actual inspection. There are two types of contours, each with specific properties and uses, in the experiment:

1. Open contour: a feature of the part, like an edge or ridge that does not form a 'closed' region. Lighting anomalies may also cause an open contour to be discovered.
2. Closed contour: a part or image feature that forms a closed region; that is, it can be followed from a specific point on the feature back to itself. A typical closed contour is a hole or the part boundary.

We are concerned with finding as many "real" contours as possible while ignoring the "false" contours. A real contour would be an actual feature of the part, while a false contour is attributed to other factors such as lighting problems (shadows, reflections) or occlusion (the probe detected as a part feature).

If we are unable to supply the machine with relatively few false contours and a majority of real contours, the actual inspection will take longer. The machine will waste time inspecting shadows, reflections, etc.

We avoid many of these problems by carefully controlling the lighting conditions of the experiment. The static environment of the manufacturing workspace allows us to provide a diffuse light source at a chosen intensity. However, simple control of the lighting is not enough. We must apply several preprocessing steps to the images before searching for contours, including:

1. Threshold the image to extract the known probe intensities.
2. Calculate the Laplacian of the Gaussian.
3. Calculate the zero-crossings of the second directional derivative.
4. Follow the "strong" zero-crossing edge responses to discover contours.

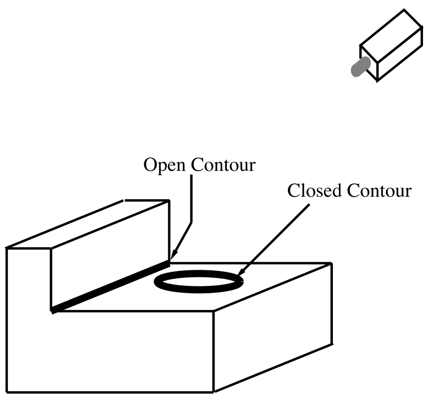


FIGURE 6.9 Edge finding in the two-dimensional image can give hints about where to look for three-dimensional features. The open contour here is generated where two faces meet. The corresponding contour is then explored by both the stereo layer and the CMM machine.

Zero-crossings

The Marr-Hildreth operator [11] is used to find areas where the gray-level intensities are changing rapidly. This is a derivative operator, which is simply

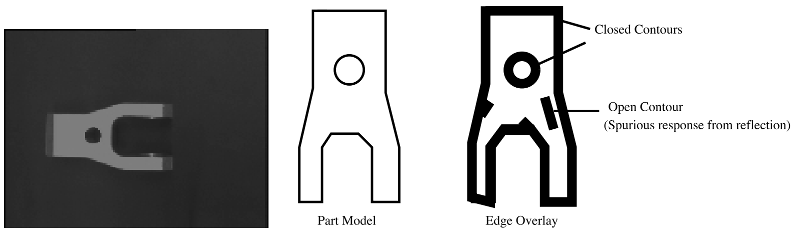


FIGURE 6.10 A contour discovery example.

the thresholded image convolved with the Laplacian of a Gaussian. The operator is given by:

$$\Delta^2 G(x, y) = \frac{1}{\sigma}\frac{x^2 + y^2}{\sigma^2} - 2e^{-\frac{x^2 + y^2}{2\sigma^2}}$$

where σ is a constant that scales the Gaussian blur over the image. For large numbers, σ acts as a low-pass filter. Smaller values retain more localized features but produce results that are more susceptible to noise. This scale can be related to an image window by:

$$\sigma = \frac{w}{2\sqrt{2}}$$

where w is the actual window size in pixels. On average, we trade more accuracy for noise and rely on a robust edge follower and the intrinsic properties of contours to eliminate noise.

The zero-crossing operator calculates orientation, magnitude, and pixel location of all edge responses. This is helpful for the contour following algorithm that uses all three pieces of information.

Contour Properties

An edge response is only considered to be a contour if it satisfies two conditions: (1) each response must exceed a previously specified minimum value, and (2) the length of each edge must exceed a previously specified minimum pixel count.

Edges are followed iteratively. An edge is followed until its response falls below the minimum or we arrive at our starting position, in which case the contour is known to be closed. If a branch in the contour is encountered, the branch location is saved and following continues. We attempt to follow all branches looking for a closed contour. Branches are considered to be part of a contour because they may represent an actual feature of the part (a crack extending from a hole, for example) and should be inspected.

Once the region contours are found, they can be used in the stereo vision correspondence problem for model construction. They are also given to the machine to help drive the actual inspection process. Some closed contours and the image in which they were found are seen in Fig. 6.11.

Visual Observation of States

The visual processor supplies the proper input signals to the DRFSM DEDS as the inspection takes place. These signals are dependent on the state of the scene and are triggered by discrete events that are observed by the camera.

The visual processor layer is made up of several filters that are applied to each image as it is captured. Several things must be determined about the scene before a signal is produced: the location of the part,

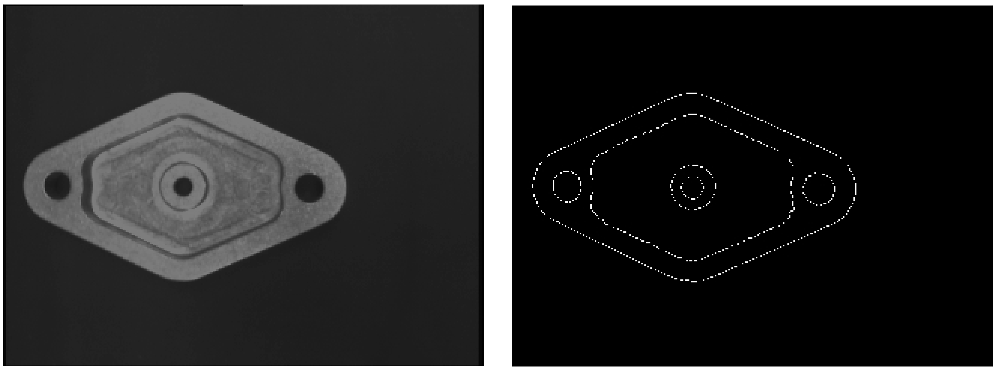


FIGURE 6.11 An image and its contours.

the location of the probe, the distance between them, the number of features on the part, and the distance to the closest feature.

First, the image is thresholded at a gray-level that optimizes the loss of background while retaining the part and probe. Next, a median filter is applied that removes small regions of noise. The image is then parsed to find all segments separated by an appropriate distance and labels them with a unique region identifier.

We are able to assume that the probe, if in the scene, will always intersect the image border. The probe tip is the farthest point on the probe region from the border. This holds true because of the geometry of the probe. An image with one region that intersects the border is the case in which the probe is touching the part.

If we have more than one region, we must discover the distance between the tip of the probe region and the part. This is done through an edge following algorithm that gives us the x, y positions of the pixels on the edge of each region. We then find the Euclidean distances between the edge points and the probe tip. The closest point found is used in producing the signal to the state machine.

Once this information is known, we are able to supply the correct signal that will drive the DRFSM DEDS. The machine will then switch states appropriately and wait for the next valid signal. This process is a recursive one, in that the machine will be applied recursively to the closed features. As the probe enters a closed region, another machine will be activated that will inspect the smaller closed region with the same strategy that was used on the enclosing region.

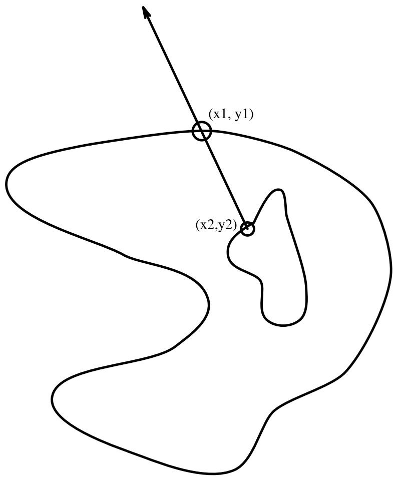


FIGURE 6.12 A closed region within another.

Deciding Feature Relationships

Having found all of the features, we now search for the relationships between them. In the final representation of intrinsic information about the part, it is important to know which feature lies "within" another closed feature.

Consider a scene with two features, a part with an external boundary and a single hole. We would like to represent this scene with the string: "C(C())". This can be interpreted as a closed region within another closed region.

To discover if feature F_2 is contained within F_1 given that we know F_1 is a closed feature, we select a point (x_2, y_2) on F_2 and another point (x_1, y_1) on F_1. Now, we project the ray that begins at (x_2, y_2) and passes through (x_1, y_1). We count the number of times that this ray intersects with F_1. If this is odd, then we can say F_2 is contained within F_1; otherwise it must lie outside of F_1 (see Figs. 6.12 and 6.13)

This algorithm will hold true as long as the ray is not tangential at the point (x_1, y_1) of feature F_1. To avoid this case, we simply generate two rays that pass through (x_2, y_2) and a neighboring pixel on F_1. If either of these have an odd number of intersections, then F_2 is contained in feature F_1.

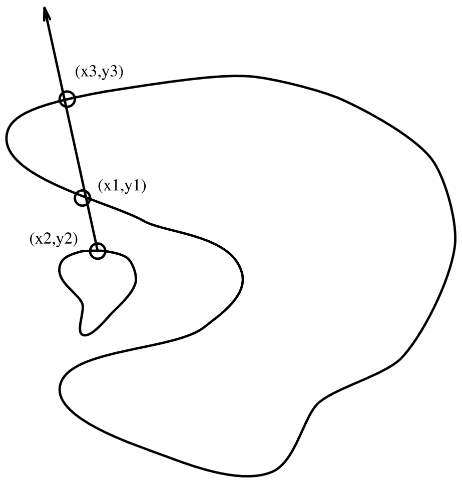


FIGURE 6.13 A closed region outside another.

An alternate method was also used to determine whether a region is inside another. A point on the contour to be checked was grown. If the grown region hit the frame, that would imply that the region is not contained; otherwise, it would be contained inside the bigger contour, and the grown region would be all the area within the bigger contour.

Knowing what features are present in the part and their relationships with each other will allow us to report the information in a string that is sent to the state machine. This process will be explained in detail in the next section.

Constructing the Recursive Relation

One of the problems encountered in this experiment was converting the set of relations between closed regions to the proposed syntax for describing objects. For example, the syntax of Fig. 6.14 is:

$$C(C(C(),C()),C())$$

and the relations generated by the image processing program are:

$$\begin{aligned} B &\in A \rightarrow (1)\\ C &\in A \rightarrow (2)\\ D &\in B \rightarrow (3)\\ D &\in A \rightarrow (4)\\ E &\in B \rightarrow (5)\\ E &\in A \rightarrow (6) \end{aligned}$$

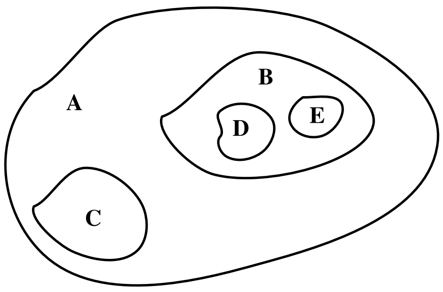


FIGURE 6.14 A hierarchy example.

These relations can be represented by a graph as shown in Fig. 6.15. The target is to convert this graph to an equivalent tree structure, which is the most convenient data structure to represent our syntax.

As a first attempt, we designed an algorithm to convert from graph representation to tree representation by scanning all possible paths in the graph and putting weights to each node according to number of visits to this node. In other words, update the depth variable of each node by traversing the tree in all possible ways and then assign the nodes the maximum depth registered from a traversal, and propagate that depth downward. Then, from these depth weights, we can remove the unnecessary arcs from the graph by keeping only the arcs that have a relation between a parent of *maximum* depth and a child, and eliminating all other parent arcs, thus yielding the required tree (Fig. 6.16).

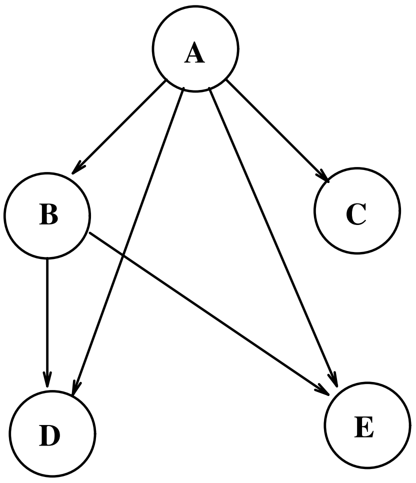


FIGURE 6.15 The graph associated with the example in Fig. 6.14.

However, we have developed a better algorithm that scans the relations; counts the number of occurrences for each closed region name mentioned in the left side of the relations, giving an array RANK(x), where $x \in \{A,B,C,\ldots\}$; and selects the relations ($x_1 \in x_2$) that satisfy the following condition:

$$\text{RANK}(x_1) - \text{RANK}(x_2) = 1$$

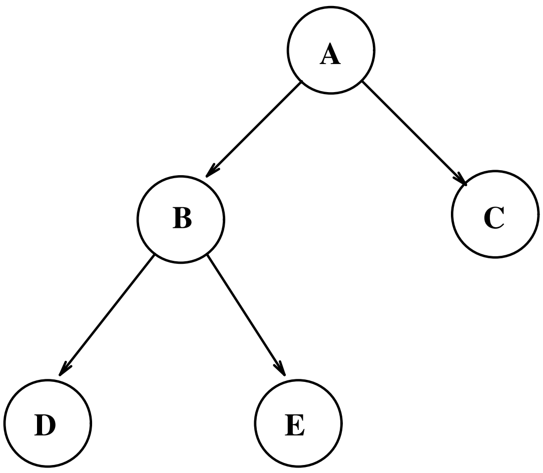


FIGURE 6.16 The tree associated with the example in Fig. 6.14.

This guarantees that all redundant relations will not be selected. The complexity of this algorithm is O((n)), where n is the number of relations. Applying this algorithm to the relations of Fig. 6.14, we obtain:

$$\begin{aligned} \text{RANK(A)} &= 0 \\ \text{RANK(B)} &= 1 \\ \text{RANK(C)} &= 1 \\ \text{RANK(D)} &= 2 \\ \text{RANK(E)} &= 2 \end{aligned}$$

The selected relations will be:

$$\begin{aligned} &\text{B} \in \text{A} \\ &\text{C} \in \text{A} \\ &\text{D} \in \text{B} \\ &\text{E} \in \text{B} \end{aligned}$$

Now, arranging these relations to construct the syntax gives:

$$\text{A(B())} \rightarrow \text{A(B(),C())} \rightarrow \text{A(B(D()),C())} \rightarrow \text{A(B(D(),E()),C())}$$

which is the required syntax. A tree representing this syntax is easily constructed and shown in Fig. 6.16. The next step would be to insert the open regions, if any, and this is done by traversing the tree from the maximum depth and upward. Any open region can be tested by checking any point in it to see whether it lies within the maximum-depth leaves of the closed regions' tree hierarchy (the test is easily done by extending a line and checking how many times it intersects a closed region, as in the test for closed regions enclosures). Then, the upper levels of the hierarchy are tested in ascending order until the root is reached or all open regions have been exhausted. Any open region found to be inside a closed one while traversing the tree is inserted in the tree as a son for that closed region. It should be noticed that this algorithm is *not* a general graph-to-tree conversion algorithm; it only works on the specific kind of graphs that the image processing module recovers. That is, the conversion algorithm is *tailored* to the visual recursion paradigm.

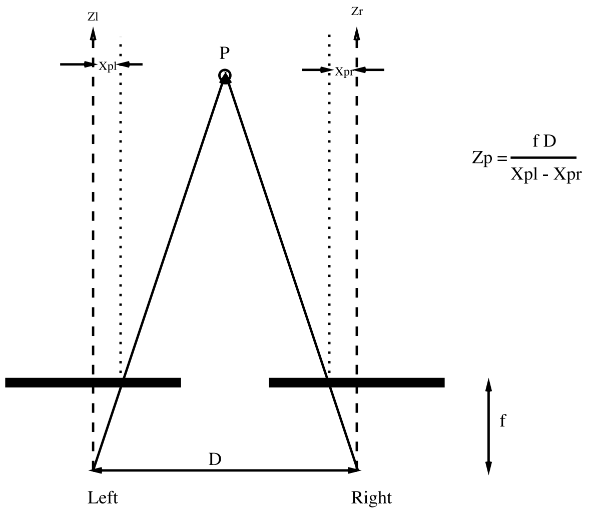


FIGURE 6.17 Pinhole camera model.

Extraction of Depth Information and World Coordinates

A crude initial model is found using stereo vision. The camera model used is a pinhole camera as shown in Fig. 6.17, corrected for radial distortion. Depths are found with such models using the disparity between feature locations in a pair of views according to the following formula:

$$Z = fD/(x_l - x_r)$$

where x_l and x_r are coordinates on the image plane, f is the focal length, and D is the disparity. Additional aspects of the camera model are discussed in the section on camera calibration.

The stereo algorithm currently in use requires no correspondence between individual features in the stereo image pair [1]. Instead, corresponding regions are found, and the disparity in their centers of mass is used for depth computation. In our experiment, closed regions are found in two images and their relationships are determined. Each closed region is described by its boundary, a contour in image coordinates. It is assumed for the initial model that these contours are planar. Given this assumption, the parameters p, q, and c of a plane must be solved for in the equation

$$Z = pX + qY + c$$

To do this, each region is split into three similar sections in both left and right images. The center of mass is computed for each section, and the system of equations solved for p, q, and c. These values are stored with the region for later output of the CAD model (we use the α_1 CAD package). It should be noted that if the centers of mass are collinear, this system will not be solvable (three non-collinear points define a plane). Also, if the centers of mass are close together, the error in discretization will cause substantial error in computation of plane parameters. In other words, if the three points are close together, an error of one pixel will cause a substantial error in the computed orientation of the plane. The effect of a one-pixel error is reduced by selecting points that are "far" apart. Thus, the technique used to split regions, determining the locations of these points, is a crucial part of the algorithm.

The most obvious and perhaps simplest technique splits contours by dividing them into three parts horizontally (see Fig. 6.18.) Because many machined features (such as holes) will produce collinear centers of mass when partitioned this way, a different technique is used. It is attempted to divide each contour into three parts of equal length (see Fig. 6.18). One region may partitioned purely by

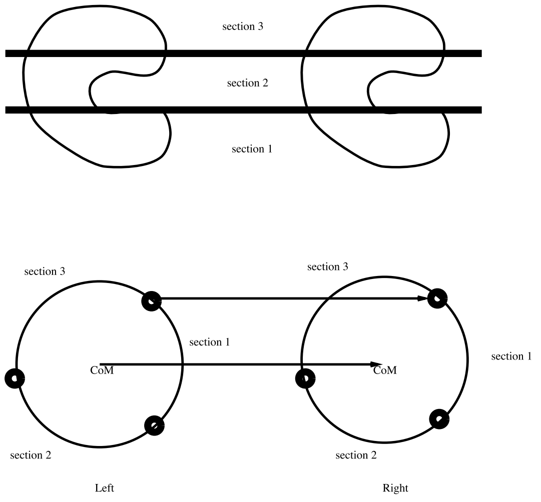


FIGURE 6.18 Region splitting algorithms.

length, but to partition the other exactly would require solution of the correspondence problem. Fortunately, the effects of error in correspondence are made minimal when averaged over a section. The first pixel in the left image's region is matched with one in the right image by translating it along a vector between the centers of mass of the regions and finding the closest pixel to this position in the right image.

In practice, this was found to work fairly well for the outermost region. However, using the same technique for smaller inner regions, error is much greater because the three centers of mass used to determine the plane parameters are closer together. A further assumption may be made, however: that the inner regions are in planes parallel to the outer region. Using this assumption, it is not necessary to split the regions into three parts, and the plane equation can be solved for c using the center of mass of the entire region. If it is assumed that all planes are parallel to the table (world x-y plane), the outermost region can be treated in like manner.

For our initial experiment, the following assumptions were made.

- The robot z axis is perpendicular to the table on which the robot is mounted.
- The table is a planar surface, parallel to the floor.
- The CCD plane of the camera is parallel to the back of the camera case.
- All object contours are in planes parallel to the table.

The experimental setup is shown in Fig. 6.19. The camera was oriented with its optical axis approximately perpendicular to the table. This was first done by visual inspection. Then, for more accuracy, a level was used on the back of the camera case and the robot tool was rotated until the camera appeared level. The robot tool frame was then recorded (as Left). This frame was used consistently to capture images for the remainder of the experiment. At that point, the problem had been constrained to finding the angle between robot x and image x. This was necessary because the stereo algorithm is based on disparity only in the image x direction.

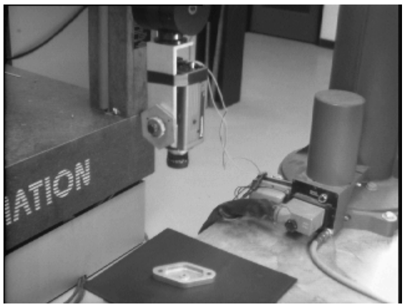


FIGURE 6.19 Experimental setup.

To accomplish the constrained motion, the following algorithm was implemented:

- Move the camera to the recorded frame.
- Take an image.
- Threshold it.
- Compute the center of mass of an object in the scene (there should be only one) in image coordinates.
- Move the camera in the robot-*x* direction.
- Take an image.
- Threshold it.
- Compute the new center of mass of an object in the scene in image coordinates.
- Compute the angle between the vector (new-old) and the image *x*-axis.
- Compute a new frame accounting for this angle and record it.
- Move to the new frame, recompute the center of mass, and display it.

At this point, the rotation part of the transform from camera coordinates to world coordinates is known, and the translational part must be determined. X- and Y-components of the translation are taken to be zero, making the world coordinate origin centered in the left frame image. The Z-component was determined by taking an image pair of a paper cut-out (thickness assumed to be zero). The Z-coordinate of this object should be the distance from the image plane to the table. This was then used to complete the homogeneous transform from camera coordinates to world coordinates:

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -1.0 & 234.1 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Several stereo image pairs were then captured using the Left and Right frames, and then used by the stereo code to produce α_1 models with the objects described in world coordinates. For a cube measuring 1 inch (25.4 mm) on a side, the resulting α_1 model was similar to a cube (lighting effects are observable), and dimensioned to 26.74 mm $\times$ 25.5 mm $\times$ 25.7 mm (h $\times$ l $\times$ w). This corresponds to percent errors of 5.2, 0.4, and 1.2, respectively. Some example images and corresponding models are shown in later sections.

Camera Calibration

Real-world cameras differ substantially from the ideal camera model typically used for discussion of stereo vision. Lens distortion, offsets from the image center, etc. are sources of error in computing range information. The camera calibration technique chosen for this project takes many of these factors into account. Developed by Roger Tsai, and implemented by Reg Willson of CMU [18, 20], this technique has been described as:

- Accurate
- Reasonably efficient
- Versatile
- Needing only off-the-shelf cameras and lenses
- Autonomous (requiring no operator control, guesses, etc.)

The technique solves for the following parameters:

- *f*-focal length
- k-lens distortion coefficient

- (Cx, Cy) image center
- Uncertainty scale factor (due to camera timing and acquisition error)
- Rotation matrix
- Translation vector

of which we use the focal length, distortion coefficient, and image center. The equations used are as follows:

$$X_u = f \cdot \frac{x}{z}$$

$$Y_u = f \cdot \frac{y}{z}$$

$$X_u = X_d \cdot (1 + k \cdot (X_d^2 + Y_d^2))$$
$$Y_u = Y_d \cdot (1 + k \cdot (X_d^2 + Y_d^2))$$

$$X_d = dx \cdot (X_f - C_x)$$
$$Y_d = dy \cdot (Y_f - C_y)$$

where *dx* and *dy* are the center-to-center distances between adjacent sensor elements on the CCD plane in the *x* and *y* directions (obtained from Panasonic); X_u and Y_u are undistorted image plane coordinates; *x*, *y*, and *z* are in the camera coordinate system; and X_d and Y_d are distorted image plane coordinates. The effects of the distortion coefficient can be seen in Figs. 6.20 to 6.23.

In the classical approach to camera calibration, computing the large number of parameters requires large-scale nonlinear search. In Tsai's method, however, the problem's dimensionality is reduced by using the radial alignment constraint to split the search into two stages [18]. In the first stage, extrinsic parameters such as Translation and Rotation parameters are found. The second solves for the intrinsic parameters (*f*, *k*, etc.).

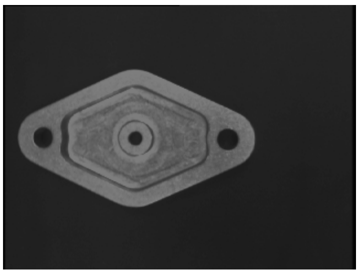


FIGURE 6.20 Raw image.

The implementation used accepts a data file containing points that are known in both image coordinates and world coordinates. For Tsai's original paper, data was obtained from a calibration grid measured with a micrometer and 400*X* microscope. For our purposes, a paper grid of 1-mm diameter dots spaced 1 cm apart was made using AutoCad and a plotter (see Fig. 6.24). A plotter was used rather than a laser printer in hopes of minimizing such errors as the stretching effect found in the output of worn laser printers. The calibration algorithm is quite sensitive to systematic errors in its input. It should be noted that for Tsai's algorithm, the camera's optical axis should be at an angle greater than 30° from the plane in which the calibration points occur. The calibration data was generated in the following manner:

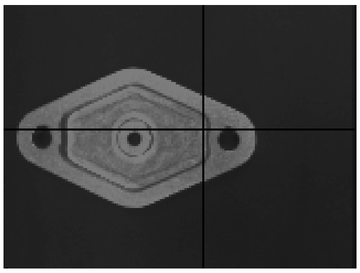


FIGURE 6.21 Corrected image, kappa = .0005.

- Capture calibration image (dot locations are known in world coordinates)
- Threshold calibration image.
- Visit each calibration image "dot" in a systematic way:
  - — Select a pixel interior to the dot.
  - — Compute the center of mass of the 8-connected region.
  - — Output the world *x-y-z*, image *x-y* information to the calibration data file.

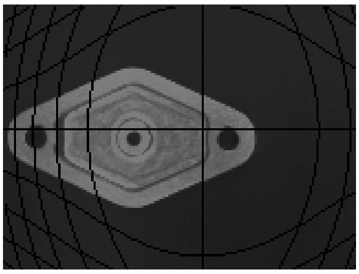


FIGURE 6.22 Corrected image, kappa = .00357 (used in our experiment).

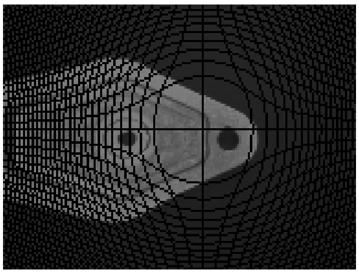


FIGURE 6.23 Corrected image, kappa = .05.

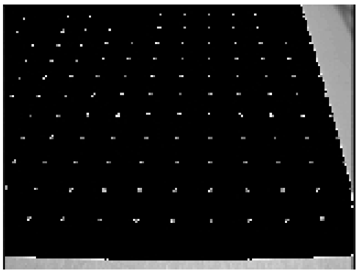


FIGURE 6.24 Thresholded calibration grid.

For our first experiment, 25 calibration points were used. Typical results are shown in Appendix B.

For future experiments, a more refined calibration data collection system may be used, possibly using the CMM as a tool to generate data points. This will facilitate outputting stereo range information in the CMM's coordinate system.

Depth Estimation Using an Illumination Table

Using stereo techniques for estimating the depths of an object's contours can be very accurate, but it is limited in that it cannot compute the depth of an occluded contour (i.e., the bottom of a hole or pocket). As shown in Fig. 6.25, the algorithm will give the depths for both contours correctly in case A, while in case B the depth of both contours will be the same.

It was attempted to solve this problem using a predefined illumination table that relates the intensity of a point on the object to the distance between this point and the camera. When the stereo algorithm detects two nested contours with the same depth, this table would be used to estimate the depth of the inner region. This method is very simple to implement, but it proved to have some drawbacks. For example, it is very sensitive to the lighting conditions, i.e., any variation in the lighting conditions will result in the invalidation of the lookup table. Also, objects being observed must have consistent surface properties. In the following section, attempts to overcome these problems are described.

Table Construction

This table is constructed off-line before running the experiment. The following assumptions were made:

- The object is formed of the same material, hence the illumination at any point is the same (assuming well distributed light and no shadows).
- The same camera with the same calibration parameters are used during the experiment.
- The lighting conditions will be the same during the experiment.

We can consider these to be valid assumptions because the manufacturing environment is totally controlled, thus, we know the object material and we set the lighting conditions as desired.

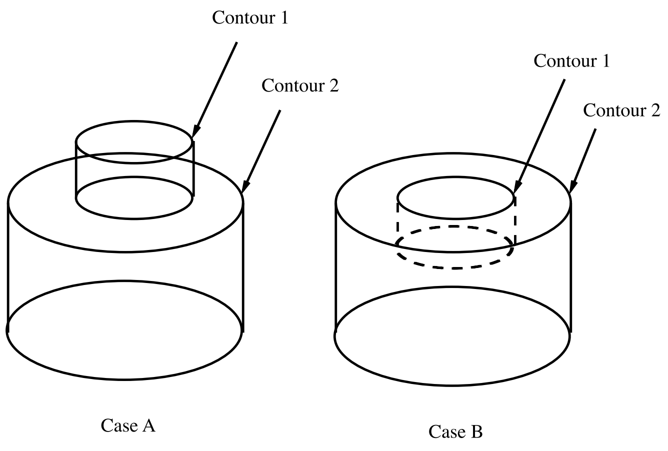


FIGURE 6.25 The problem when using stereo in depth estimate.

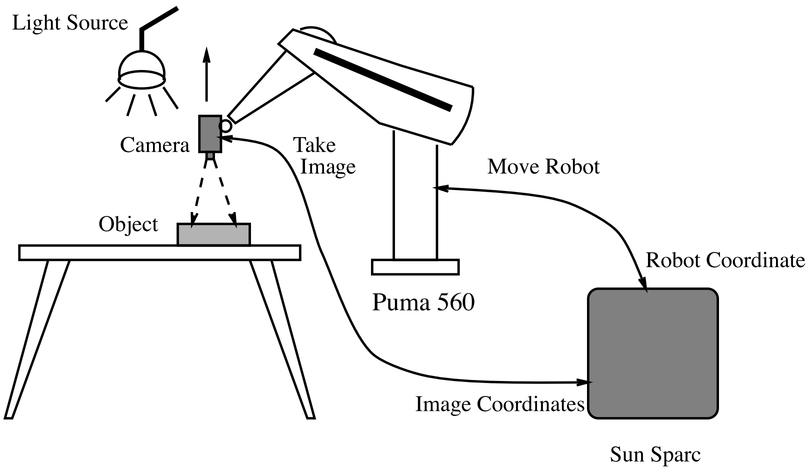


FIGURE 6.26 Constructing the illumination table off-line.

This table will be constructed only once, and will then be used for all our experiments, as long as they satisfy our assumptions. However, if we wish to examine an object with different materials, or we want to change the lighting conditions, we will have to construct a new table using the new object and the new lighting conditions. To construct this table, the robot arm that holds the camera is moved vertically in incremental steps, according to the required accuracy. At each increment, an image is taken and the intensity at the center of the image is measured; see Fig. 6.26 for the experimental setup.

Modifications

The initial implementation of this method did not produce the expected results because of the noise in the images taken at each depth. Several enhancements were added to this method to reduce the effect of noise. First, instead of measuring the intensity at one point, we take the average of the intensities of a set of points that constitutes a rectangular window. By changing the window size, we can control the smoothing degree of the measured intensity. The second enhancement is also based on averaging, by taking several images at each height and taking the average of the calculated average window intensities. After applying these two modifications, the effect of noise was greatly reduced.

Another modification was to move the light source with the camera to increase the difference in the measured intensity at each height, which, in turn, should have increased the resolution of our table.

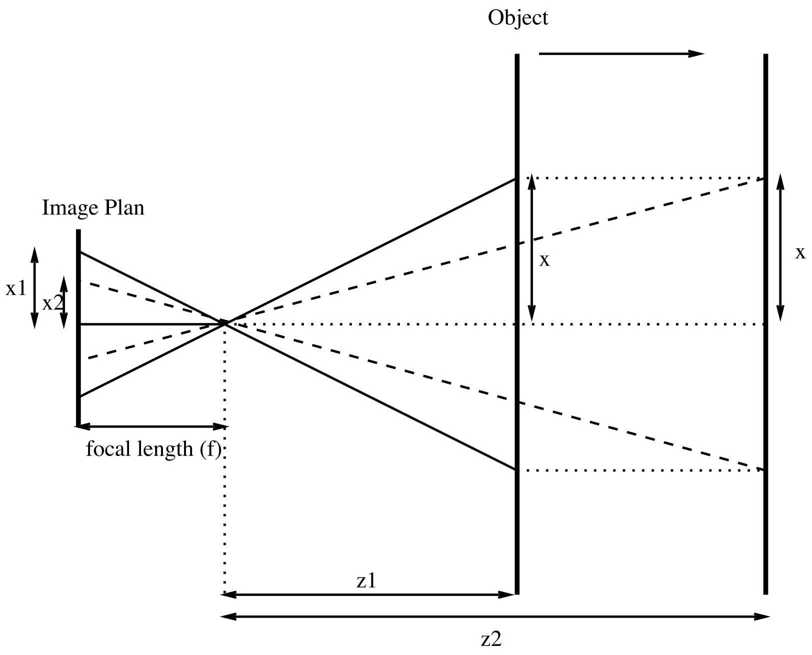


FIGURE 6.27 Changing window size assuming pinhole camera model.

One last enhancement was incorporated based on the perspective-projective transform from world points to image points. The window size used to calculate the average intensity at each height should be the same; but according to the image formation equations using the pinhole camera model, the corresponding window in the image will change according to the distance between the camera (image plane) and the object. From Fig. 6.27, using simple trigonometry, we obtain the following relation between the image window size and the distance z between the object and the camera:

$$\frac{x1}{x2} = \frac{z2}{z1}$$

which shows that the image window size is inversely proportional to the distance between the object and the camera. Thus, we must calculate the new window size at each height, which will be the number of pixels used for averaging.

Figure 6.28 shows a graph for the constructed illumination table used in our experiment. It shows that the intensity decreases when the distance between the object and the camera increases, but it also shows that any change in the lighting condition will give different results for the illumination table.

This method also has some pitfalls. First, it is very sensitive to any light change, as shown in Fig. 6.28. Second, the difference in illumination values for two close depths is very small. For example, in our experiment, the total range of differences within 10 cm was less than 30 gray-levels. Finally, it still has small amounts of noise at some points. We are now developing another method for determining depth from focus. This method involves calculating distances to points in an observed scene by modeling the effect that the camera's focal parameters have on images acquired with a small depth of field [7].

6.5 Sensing to CAD Interface

An important step in the reverse-engineering process is the accurate description of the real-world object. We generate an α_1 model from a combination of three types of scene information:

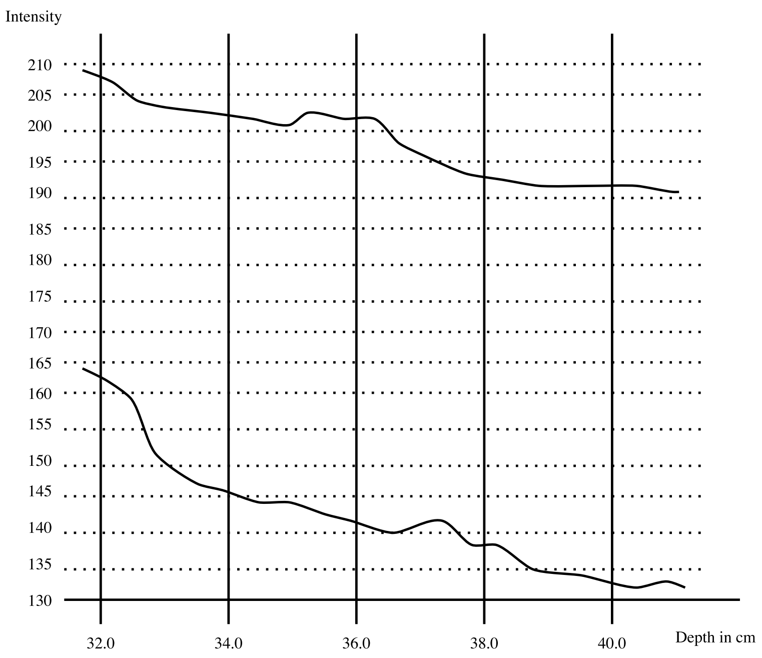


FIGURE 6.28 Two different results when changing the lighting conditions.

- Two-dimensional images and feature contours
- Stereo vision depth information
- Touch information from the CMM (still to be implemented)

Using each sensing method, we are able to gather enough data to construct the accurate CAD model necessary for reverse engineering (see Fig. 6.29.) The two-dimensional images provide feature detection that is in turn used by the stereo system to build feature models. Finally, the CMM eliminates uncertainty through the exploration of the features.

The state machine and the sensors are able to produce a set of data points and the respective enclosure relationships. Each feature is constructed in α_1 independently, and the final model is a combination of these features. This combination is performed using the recursive structure of the object by forming the corresponding string for that object and generating the code by parsing this string recursively. The third dimension is retrieved from the stereo information and the illumination table as described previously. An example for a reconstructed part is shown in Fig. 6.30.

This interface is one of the most important modules in this work, because it is the real link between inspection and reverse engineering. We have chosen α_1 as the CAD language because it has very powerful features, in addition to the fact that it has interfaces with some manufacturing machines, which allows us to actually manufacture a hard copy of the part. This will be our next step, so that the output of our next experiment will be another part, hopefully identical to the original part.

Contours to Splines

In the initial stage of our sensing to CAD interface, we translated the ranged contours we found into spline curves.

Both closed and open contours are represented as ordered sets of points. The contour points are used as control points on a spline curve in the α_1 system. It is important not to use all of the contour points while fitting the spline. In many cases, there are more than a thousand points in the original image. This gives an over-constrained solution.

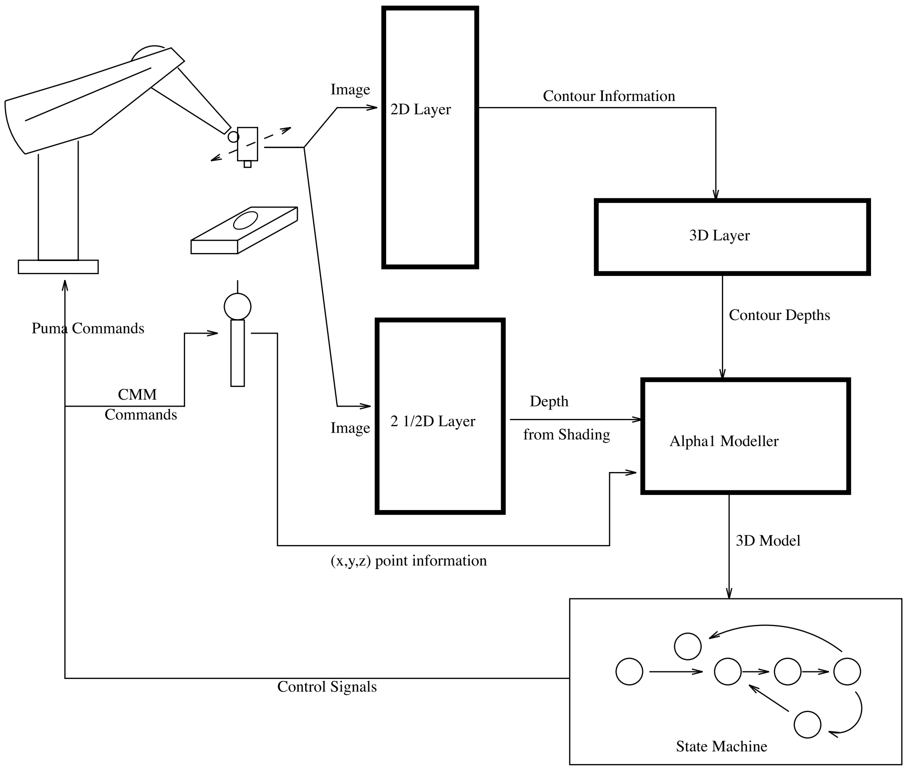


FIGURE 6.29 The role of an internal CAD model in the inspection machine.

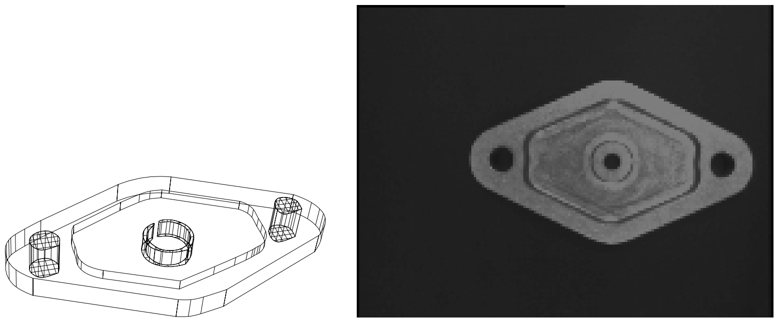


FIGURE 6.30 A rough α_1 surface model extracted from the machine vision.

Thinning Contours

We must supply a list of control points to α_1 that will fit a spline accurately to the original real-world feature. Therefore, we must decide which points contribute to the actual shape of the feature and which points are simply redundant.

Obviously, regions of high curvature are important to the overall shape of the feature, while low curvature regions will not play as important a role. We fit lines to each contour and represent them as polyline segments in α_1. Each line only consists of its endpoints rather than all the image points along its length. All of the line segments and splines that make up a particular contour are combined together using the α_1 profile curve.

An example is shown in Fig. 6.31. The straight lines in this closed contour are found, and the corner points are used as "important" points to the α_1 model. Points along the top arc are all used so that a spline can be fit to them accurately. The final region is represented as the combination of the lines and curves that makes up its length.

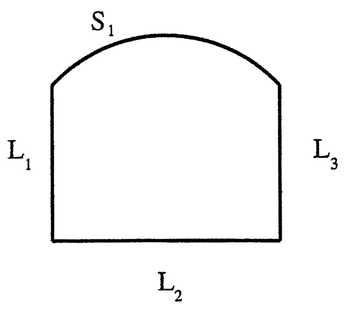


FIGURE 6.31 L_1, L_2, and L_3 are fit to lines before they are used in α_1. S_1 has too much curvature and all of its points are used to describe a piecewise spline.

Contours to Machined Features

Although the lines and splines representation proved useful, another representation was found to be needed to describe our machined parts. A set of machinable features, as implemented in the α_1 modeling system [2], has been selected for this representation. This set includes profileSides (for describing the outside curve), profilePockets (for describing interior features generated by milling), and holes (for describing features generated by drilling). Although not within the scope of current research, the method being described is extensible to include other features, such as slots, bosses, etc. Using this subset, most parts which meet the requirements of the visual processing algorithms can be transformed to a machinable α_1 representation.

The Algorithm

The transformation algorithm is as follows:

- Input: raw images, part string, and ranged contour representation
- Convert part string to tree representation (see Fig. 6.32)
- Generate stock using bounding box (see Fig. 6.33)
- Generate profileSide for outermost contour (see Fig. 6.34)
- Class each subfeature as positive or negative relative to its predecessors (see Fig. 6.35)
- Recursively descend the part tree, starting with the outermost contour's children:
  - If negative, check for positive subfeatures relative to it (if there are none, produce a hole or profile pocket depending on curvature; otherwise, there must be a profile pocket with an island; produce both)
  - Otherwise, check to see if this island needs to be trimmed
- Output: *α*_1 model composed of machinable features

Note that this algorithm assumes that the outermost contour is the highest. This limitation can be overcome by a simple check at the start, and subsequent treatment of the outer contour as a feature within a blockStock feature.

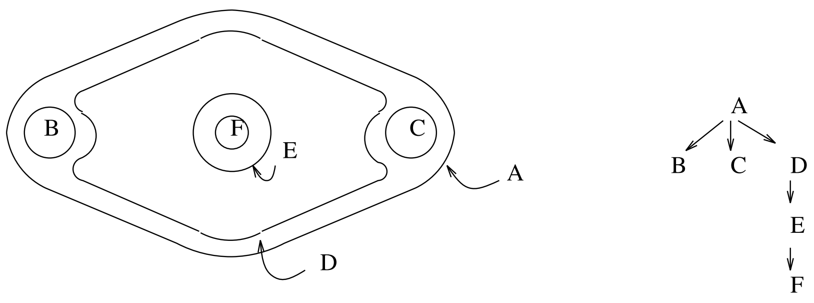


FIGURE 6.32 Sample tree representation.

Data Points to Arcs and Lines

Contours are converted to holes or α_1 profile curves, defined by combinations of arcs and lines. This is accomplished using the curvature along the contour. Areas in which the curvature is zero (or below a small threshold) for a specified distance along the contour are considered to be line segments. Areas in which the curvature is constant for a specified distance are considered to be arcs. Curvature, k, at a point on a curve is defined to be the instantaneous rate of change of the curve's slope, ϕ, with respect to curve length, s [15]:

$$k(s) = d\phi(s)/ds$$

where

$$ds = \sqrt{dx^2 + dy^2}$$

Slope is taken to be the orientation of the gradient of the difference of Gaussians (DOG) function. The DOG function is an approximation to the Laplacian as mentioned in the Zero-Crossings section of this chapter. The derivatives of slope are computed using a forward difference technique, and the results are smoothed a user-controlled number of times. A graph of curvature vs. distance along a curve can be seen in Fig. 6.36. For each arc segment, a circle is fitusing a least-squares fit [14], and then the endpoints of the arc segment are grown until the distance from the contour to the fitted circle exceeds a tolerance. This process is repeated until growing has no effect or another segment is reached. A similar method is used for the line segments. Segment data is stored as a linked list (see Fig. 6.37).

A Hough transform technique was considered for fitting curves. Although very easy to implement (one was implemented in about an hour for comparison), it was found to be too expensive in terms of memory. See Appendix C for a comparison between the technique used and the Hough transform.

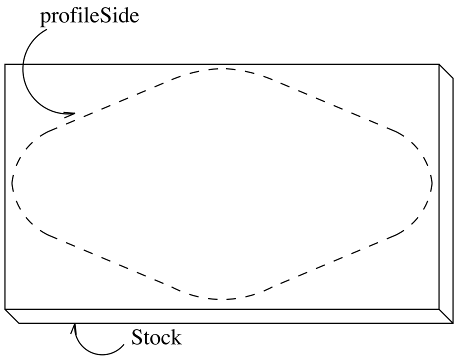


FIGURE 6.33 Stock and profileSide.

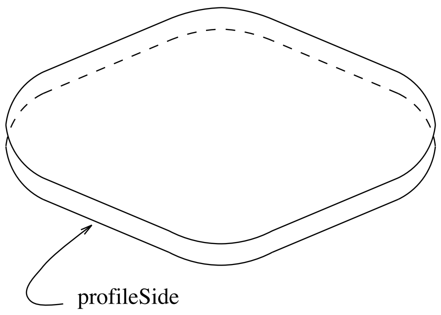


FIGURE 6.34 ProfileSide.

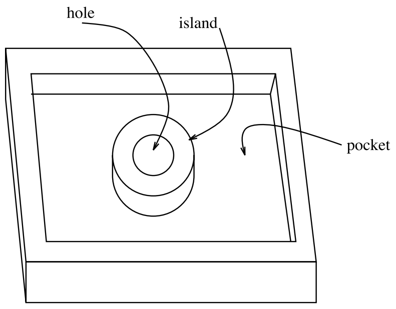
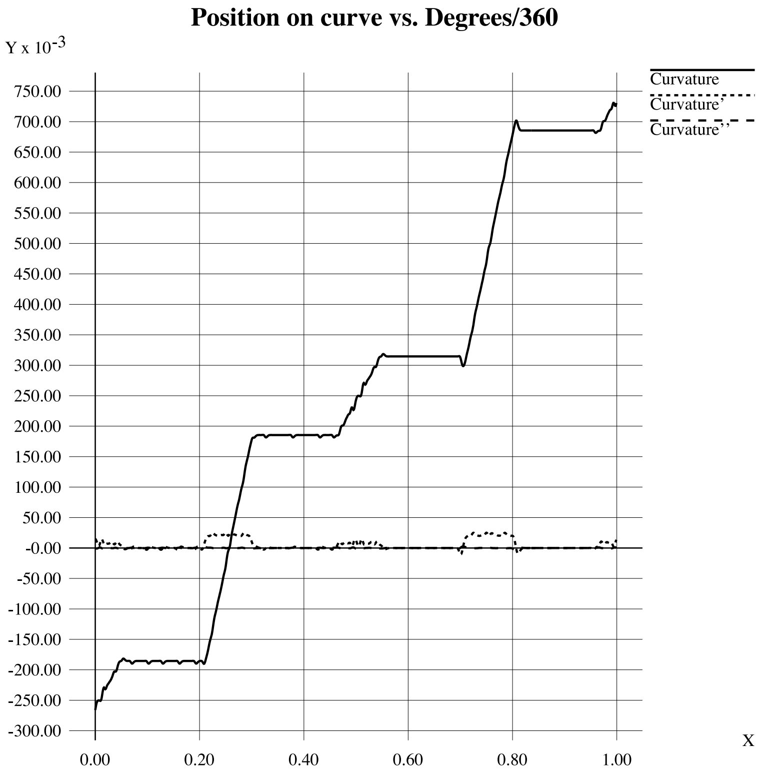


FIGURE 6.35 Positive (island) and negative (hole and pocket) features.

Arcs and Lines to Machined Features

If a negative feature contains a positive feature, then it must be a profilePocket with an island (the positive feature). The island is trimmed to the height of the positive feature with a profileGroove. If the negative feature contains no positive feature and is composed of only one arc segment, then it can be represented by a hole. To be a hole, the arc's radius must match one of a list of drill sizes within a tolerance. If a hole contains no other features, and the interior of the raw image is below a threshold, it can be considered to be a through-hole.



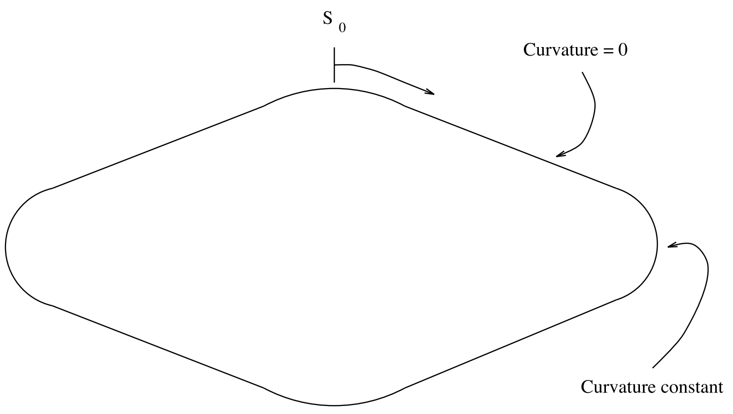


FIGURE 6.36 Curvature, and first and second derivatives.

Some aspects of machined features are difficult to measure accurately using our current image processing algorithms. For example, fillets between line segments in a profilePocket may not be within the accuracy of our vision, but are necessary for machineability. In cases such as these, default values are selected so as to have minimal deviation from the reverse-engineered model, yet allow the model to be machined. It is anticipated that some aspects (such as chamfers and threads) may be detected, although not accurately measured with vision.

In its current implementation, each contour is treated as, at most, one machined feature (some contours may be islands and therefore part of another contour's feature). Future work will allow contours to be made from multiple features if appropriate. For example, combinations of drilled

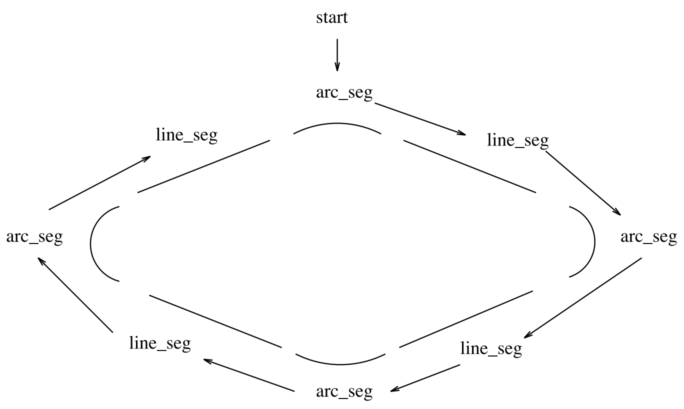


FIGURE 6.37 Contour segment data structure.

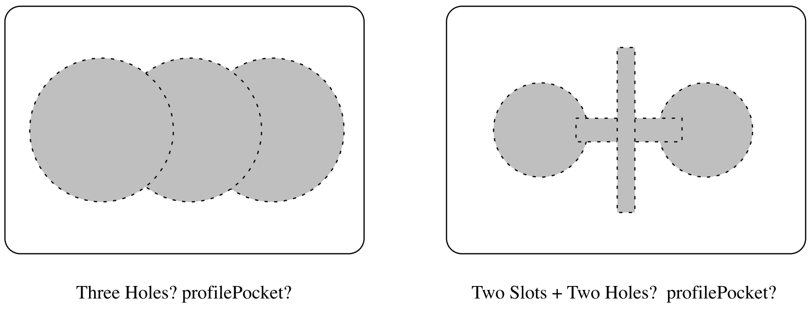

Three Holes? profilePocket?
Two Slots + Two Holes? profilePocket?

FIGURE 6.38 Possible combinations.

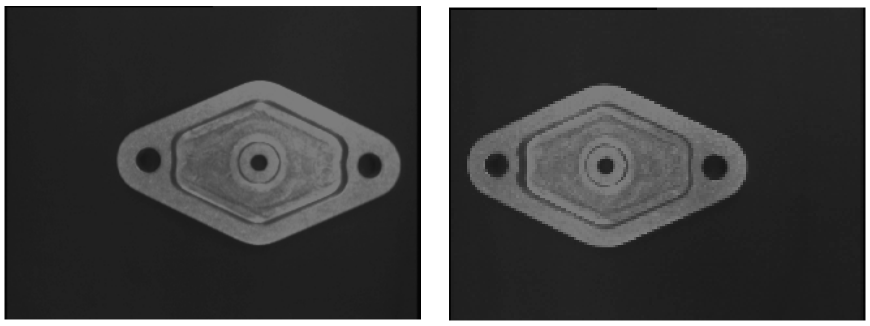


FIGURE 6.39 Stereo image pair from which ranged contours are computed.

holes, slots, and pockets may be produced (see Fig. 6.38), based on a machining/inspection time/cost analysis, which will include such factors as time needed to select and load tools (operator), change tools, etc. This problem has some characteristics that may be best solved through artificial intelligence or optimization techniques.

Results

Although the model at this intermediate stage is still crude, it was considered a useful test to have a part manufactured from it. This intermediate stage model will later be updated with CMM data as described in the section on Integration Efforts.

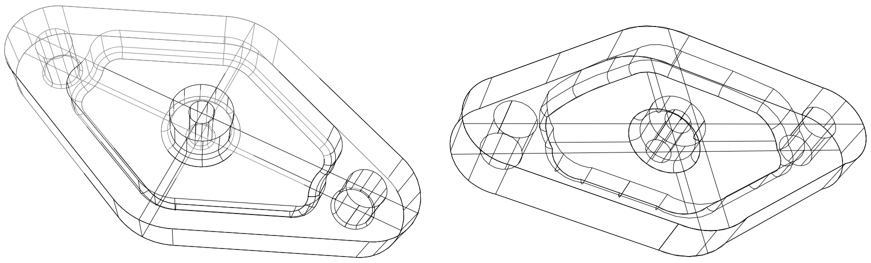


FIGURE 6.40 Original and reverse-engineered part models.

The right portion of Fig. 6.40 shows the result of applying this method to an actual part. The original part model is shown in the left portion of that figure and the stereo image pair used in image processing is shown in Fig. 6.39. Note that although the original CAD model was available in this case, it is used only for demonstration purposes, not as part of the reverse-engineering process. The reverse-engineered model was used to create a process plan and machine a part on the Monarch VMC-45 milling machine. The original part and reproduction can be seen in Fig. 6.41.

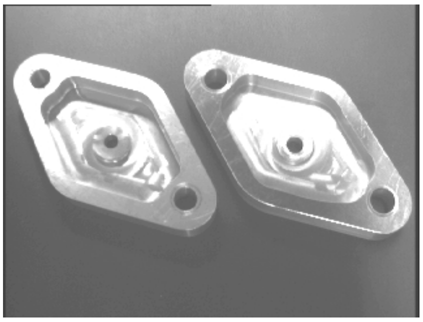


FIGURE 6.41 Original and reproduction.

6.6 System Integration

The Initiation Experiment

This experiment was performed to integrate the visual system with the state machine. An appropriate DRFSM was generated by observing the part and generating the feature information. A mechanical part was put on a black velvet background on top of the coordinate measuring machine table to simplify the vision algorithms. The camera was placed on a stationary tripod at the base of the table so that the part was always in view. The probe could then extend into the field of view and come into contact with the part, as shown in Fig. 6.42.

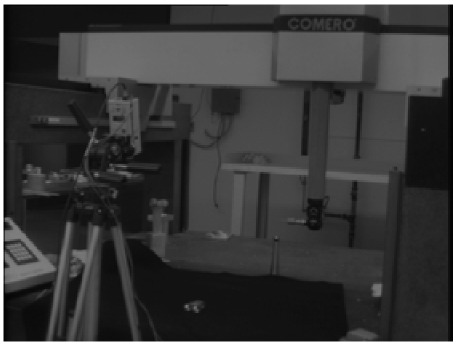


FIGURE 6.42 Experimental setup.

Once the first level of the DRFSM was created, the experiment proceeded as follows. First, an image was captured from the camera. Next, the appropriate image processing takes place to find the position of the part, the number of features observed (and the recursive string), and the location of the probe. A program using this information produces a state signal that is appropriate for the scene. The signal is read by the state machine and the next state is produced and reported. Each closed feature is treated as a recursive problem; as the probe enters a closed region, a new level of the DRFSM is generated with a new transition vector. This new level then drives the inspection for the current closed region.

The specific dynamic recursive DEDS automaton generated for the test was a state machine G (shown in Fig. 6.43.); where the set of states $X = \{\text{Initial, EOF, Error, A, B, C, D}\}$ and the set of transitional events $\Sigma = \{1, 2, 3, 4, 5, 6, 7, 8, 9, \text{eof}\}$. The state transitions were controlled by the input signals supplied by intermediate vision programs. There are four stable states A, B, C, and D that describe the state of the

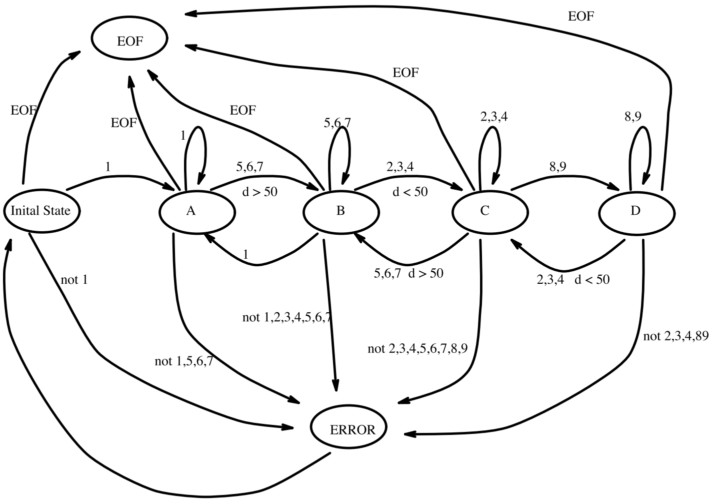


FIGURE 6.43 State machine used in test.

probe and part in the scene. The three other states—Initial, Error, and EOF—specify the actual state of the system in special cases. The states can be interpreted as:

Initial State: waiting for first input signal

A: part alone in scene

B: probe and part in scene, probe is far from part

C: probe and part in scene, probe is close to part

D: probe touching or overlapping part (recursive state)

Error: an invalid signal was received

EOF: the end of file signal was received

The Second Experiment

In the second experiment, we use a robot arm (a PUMA 560), a vision sensor (B/W CCD camera) mounted on the end effector, and a probe to simulate the coordinate measuring machine (CMM) probe, until the necessary software interface for the CMM is developed. There are several software interfaces on a Sun Sparcstation, for controlling all these devices (see Fig. 6.44.)

A DRFSM DEDS algorithm is used to coordinate the movement of the robot sensor and the probe. Feedback is provided to the robot arm, based on visual observations, so that the object under consideration can be explored. This DRFSM was generated by GIJoe, as shown in Fig. 6.6. The DEDS control algorithm will also guide the probe to the relevant parts of the objects that need to be explored in more detail (curves, holes, complex structures, etc.) Thus, the DEDS controller will be able to *model, report,* and *guide* the robot and the probe to reposition *intelligently* in order to recover the structure and shape parameters. The data and parameters derived from the sensing agent are fed into the CAD system for designing the geometry of the part under inspection. We used the α_1 design environment for that purpose. Using the automatic programming interface we have developed for α_1, we generate the required code to reconstruct the object using the data obtained by the sensing module.

Running the Experiment

The first step in running this experiment is setting the lighting conditions as desired (same conditions when constructing the reflectance map table), then initializing the robot and the camera and set them to initial positions. The experiment starts by taking images for the object from two positions, to

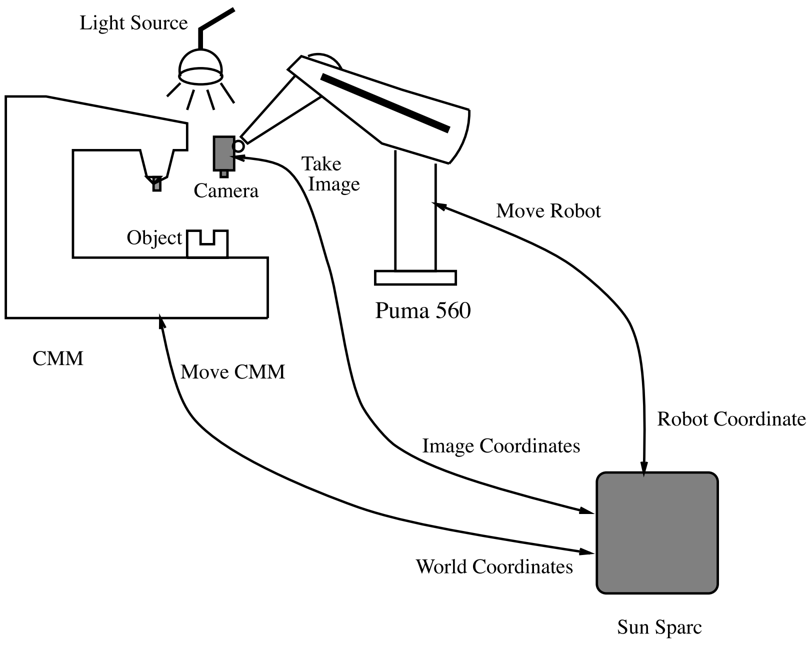


FIGURE 6.44 An experiment for inspection and reverse engineering.

generate two sets of contours to be fed into the stereo module for depth estimation. Using the stereo module with the assistance of the reflectance map table and the camera calibration module, an initial set of world coordinates for these contours is generated. Next, the DRFSM DEDS machine drives the probe and the robot arm holding the camera to inspect the object using the information generated from the stereo module and the relation between the object's contours. Figure 6.45 shows the DRFSM for this experiment.

This machine has the following states:

- **A:** the initial state, waiting for the probe to appear.
- **B:** the probe appears, and waiting for it to close. Here, close is a relative measure of the distance between the probe and the current feature, since it depends on the level of the recursive structure. For example, the distance at the first level, which represents the outer contours or features, is larger than that of the lower levels.
- **C:** probe is close, but not on feature.
- **D:** the probe appears to be on feature in the image, and waiting for physical touch indicated from the CMM machine.
- **E:** (the recursive state) physical touch has happened. If the current feature represents a closed region, the machine goes one level deeper to get the inner features by a recursive call to the initial state after changing the variable transition parameters. If the current feature was an open region, then the machine finds any other features in the same level.
- **F:** this state is to solve any vision problem happens during the experiment. For example, if the probe is occluding one of the features, then the camera position can be changed to solve this problem.
- **ERROR 1:** usually, there is time limit for each part of this experiment to be done. If for any reason, one of the modules does not finish in time, the machine will go to this state, which will report the error and terminate the experiment.

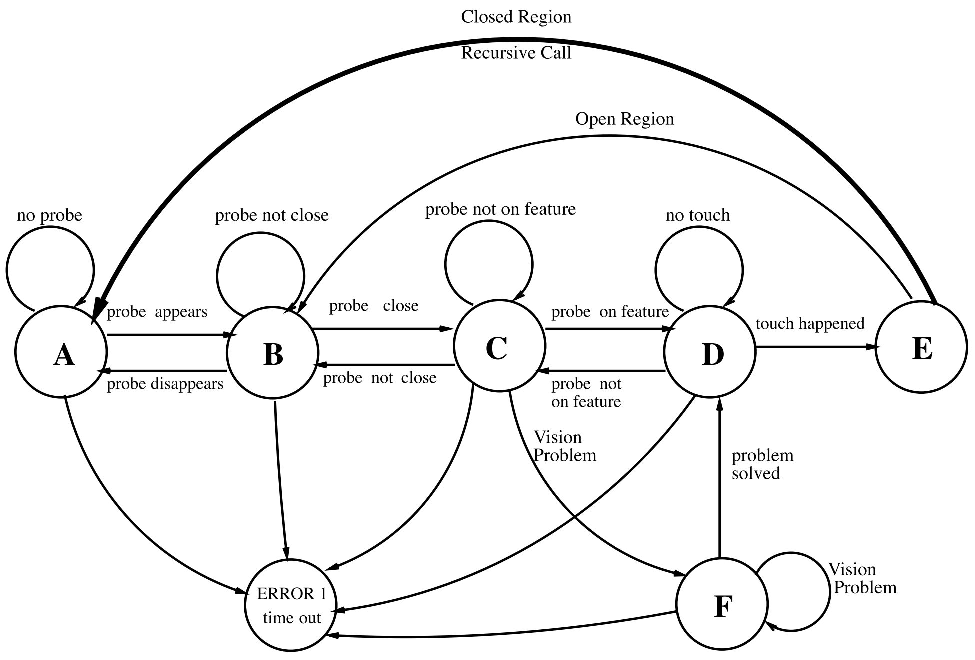


FIGURE 6.45 The DRFSM used in the second experiment.

A set of final world coordinates for the contours is obtained and fed to the α_1 interface, which in turn generates the required code for generating an α_1 model for the the object. Figure 6.46 shows a block diagram for this experiment with the results after each step.

Experimental Results, Automated Inspection

Two machine parts were used in the experiment to test the inspection automaton. The pieces were placed on the inspection table within view of the camera. Lighting in the room was adjusted so as to eliminate reflection and shadows on the part to be inspected.

Control signals that were generated by the DRFSM were converted to simple English commands and displayed to a human operator so that the simulated probe could be moved.

The machine was brought online and execution begun in State A, the start state. The camera moved to capture both 2D and 3D stereo vision information and a rough α_1 model was constructed to describe the surface, as shown in Fig. 6.47. The reconstruction takes place in state A of the machine. The constructed model is used by the machine in subsequent states. For example, the distance between the probe and the part is computed using this model and the observed probe location.

After initiating the inspection process, the DRFSM transitioned through states until the probe reached the feature boundary. The state machine then called for the closed region to be recursively inspected until finally, the closed regions are explored and the machine exits cleanly. Figures 6.48, 6.49, and 6.50 depict some exploration sequences.

6.7 Summary of Current Developments

This summary concludes the chapter by outlining some of the goals and progress within the project. We first describe some goals and methodology, and then outline current and past activities.

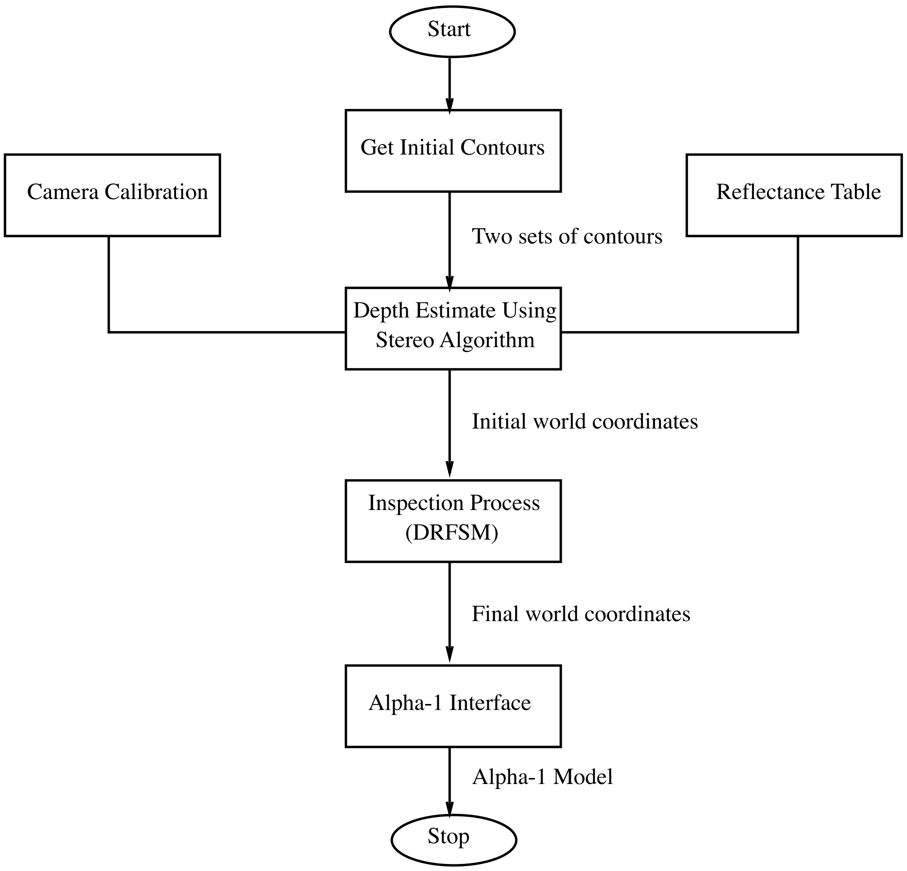


FIGURE 6.46 Block diagram for the experiment.

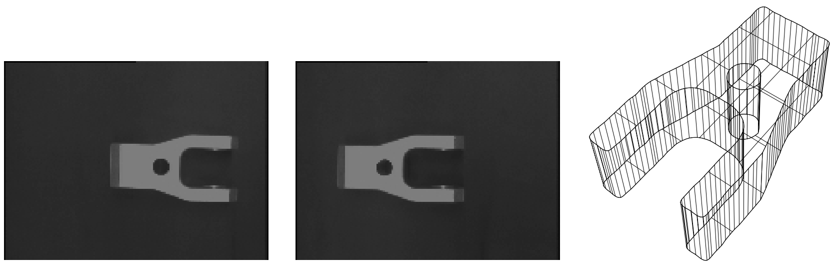


FIGURE 6.47 The two stereo images and the final α_1 model that was found in the experiment.

Goals and Methodology

We use an observer agent with some sensing capabilities (vision and touch) to actively gather data (measurements) of mechanical parts. Geometric descriptions of the objects under analysis are generated and expressed in terms of a CAD System. The geometric design is then used to construct a prototype of the object. The manufactured prototypes are then to be inspected and compared with the original object using the sensing interface and refinements made as necessary.

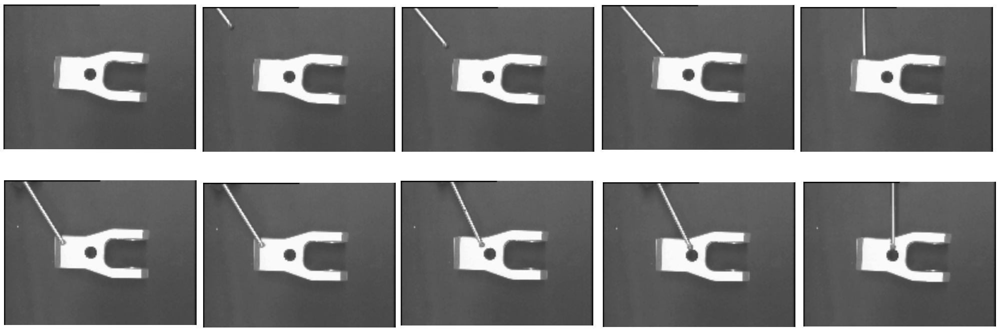


FIGURE 6.48 Test sequence (1).

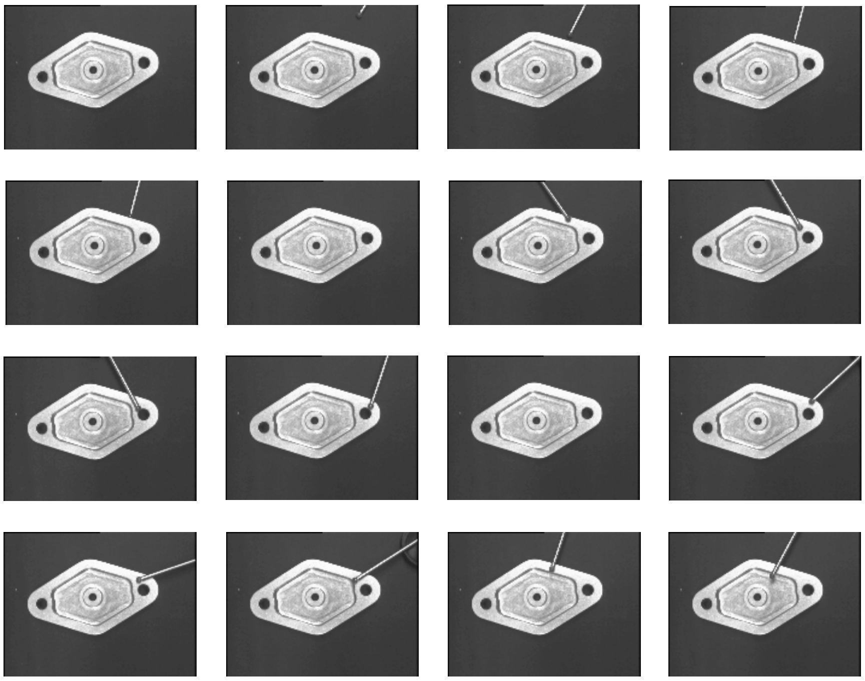


FIGURE 6.49 Test sequence (2).

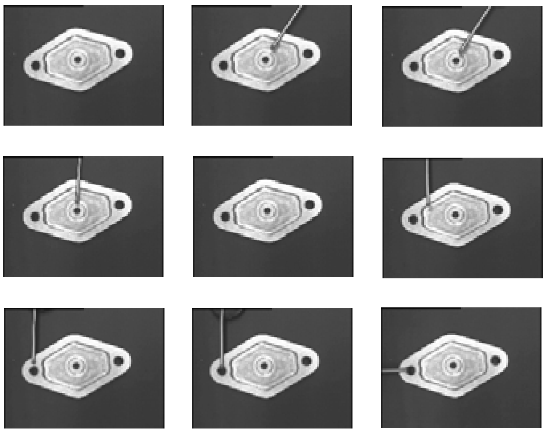


FIGURE 6.50 Test sequence (2) (contd.).

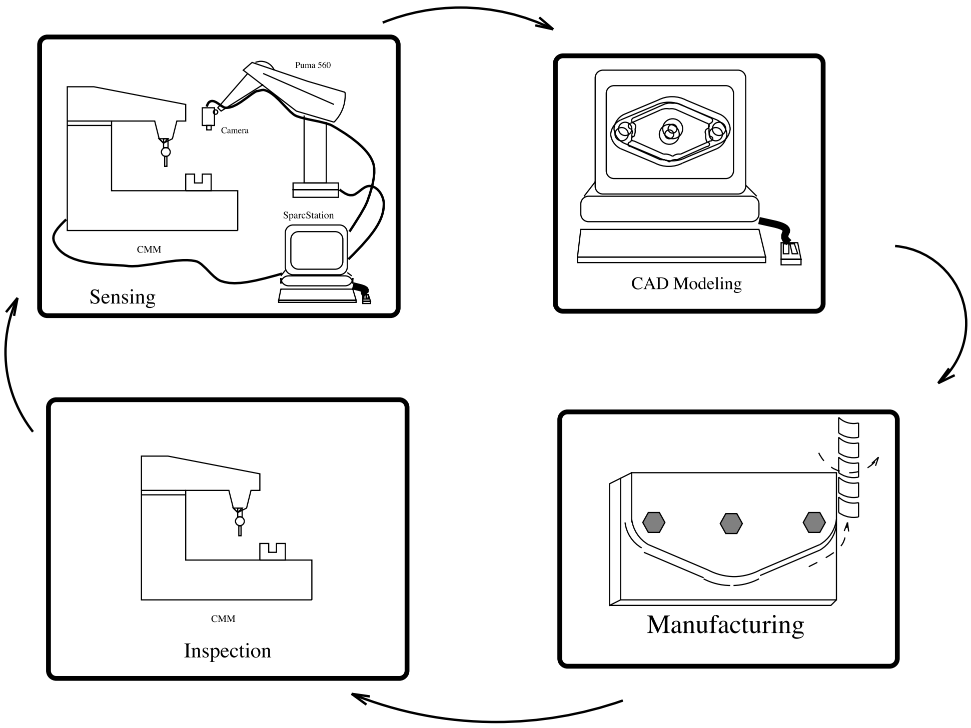


FIGURE 6.51 Closed-loop reverse engineering.

The application environment we are developing consists of three major working elements: the sensing, design, and manufacturing modules. The ultimate goal is to establish a computational framework that is capable of deriving designs for machine parts or objects, inspect and refine them, while creating a flexible and consistent engineering environment that is extensible. The control flow is from the sensing module to the design module, and then to the manufacturing component. Feedback can be re-supplied to the sensing agent to inspect manufactured parts, compare them to the originals, and continue the flow in the loop until a certain tolerance is met (see Fig. 6.51). The system is intended to be ultimately as autonomous as possible. We study what parts of the system can be implemented in hardware. Some parts seem to be inherently suited to hardware, while other parts of the system it may be possible to put in hardware; but experimentation will provide the basis for making that decision. Providing language interfaces between the different components in the inspection and reverse-engineering control loop is an integral part of the project.

Current Developments

We use a robot arm (a PUMA 560), a vision sensor (B/W CCD camera) mounted on the end effector, and will be using the coordinate measuring machine (CMM) with the necessary software interfaces to a Sun SparcStation as the sensing devices. A DRFSM DEDS algorithm is used to coordinate the movement of the robot sensor and the CMM. Feedback is provided to the robot arm, based on visual observations, so that the object(s) under consideration can be explored. The DEDS control algorithm will also guide the CMM to the relevant parts of the objects that need to be explored in more detail (curves, holes, complex structures, etc). Thus, the DEDS controller will be able to *model, report,* and *guide* the robot and the CMM to reposition *intelligently* in order to recover the structure and shape parameters.

The data and parameters derived from the sensing agent are then fed into the CAD system for designing the geometry of the part(s) under inspection. We use the α_1 design environment for that purpose. The goal is to provide automatic programming interfaces from the data obtained in the sensing module to the α_1 programming environment. The parametric and 3-D point descriptions are to be integrated to provide consistent and efficient surface descriptions for the CAD tool. For pure inspection purposes, the computer aided geometric description of parts could be used as a *driver* for guiding both the robotic manipulator and the coordinate measuring machine for exploring the object and recognizing discrepancies between the real part and the model. The computer aided design parameters will then to be used for manufacturing the prototypes.

The software and hardware requirements of the environment are the backbone for this project. We selected parts of the system for possible hardware implementation. The DEDS model, as an automaton controller, is very suitable for Path Programmable Logic (PPL) implementation. A number of the visual sensing algorithms could be successfully implemented in PPL, saving considerable computing time. There is a lot of interfacing involved in constructing the inspection and reverse-engineering environments under consideration. Using multi-language object-based communication and control methodology between the three major components (sensing, CAD, and CAM) is essential.

Past, Current, and Future Activities

Completed Activities

- Designed the DRFSM DEDS framework for recursive inspection
- Implemented image processing modules for recognizing features and probe position on the parts
- Designed and implemented visual structure recovery techniques for machine parts (using stereo, contour, and illumination map data) and implemented calibration routines
- Designed and implemented a sensing to CAD interface for generating α_1 code for bodies from depth, contour (and data reduction) illumination map, and the recursive feature relationships
- Implemented the DRFSM DEDS automata for recursive inspection (using robot-held camera, probe, and actual parts)
- Designed sensor and strategy-based uncertainty modeling techniques for the robot-held camera, for recovering the DEDS transitional "events" with uncertainty
- Designed and implemented a modification to an existing reactive behavior design tool (GIJoe) to accommodate "dumping" the code of DRFSM DEDS from a graphical interface (used to draw the inspection control automaton)
- Implemented feature identification for subsequent manufacturing (from sensed data, i.e., what does set(s) of sensed data points "mean" in terms of manufacturing features)
- Manufactured parts from camera reconstructed α_1 surfaces

Current Activities

- Designing the DEDS to VLSI design language interface (a graphical interface)
- Designing and implementing the software "uncertainty" module for subsequent hard-wiring into a chip
- Using focusing, motion, moments, shading, and more accurate robot and camera calibration techniques to enhance the visual processing
- Feature *interaction* identification for manufacturing (i.e., how can sensed features best be represented for manufacturing)

- Modifying the sensing to CAD interface for allowing CMM sensed data, in addition to visual data
- Implementing the DRFSM DEDS automata for recursive inspection and reverse engineering (using moving camera, CMM and actual parts)
- Implementing "safety" recursive DEDS for checking the sensing activities, for example, positions of probe, part, and camera

Future Activities

- Implement the VLSI modules for the DRFSM DEDS controller
- Implement the "uncertainty" chip
- Manufacture parts from camera and CMM reconstructed α_1 surfaces (with feature interaction identification built in)
- Writing and using a common shared database for storing data about the geometric models and the rules specifying the communication between the different phases
- Implement sensor-based noise modeling modules for the robot-held camera and the CMM (hardware and software)

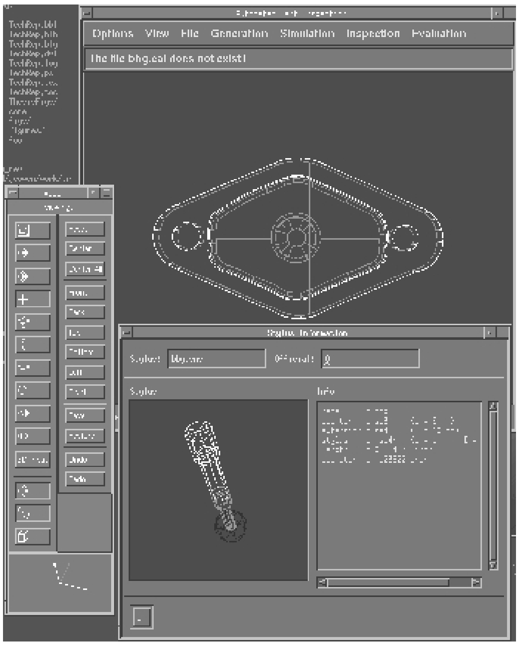
6.8 Integration Efforts

The following explains some of the integration efforts within the different areas of the project.

Robotics and Sensing

We intend to develop a software interface for the CMM machine, and a discrete event dynamic system (DEDS) algorithm will be used to coordinate the movement of the robot sensor and the CMM. The DEDS control algorithm will also guide the CMM to the relevant parts of the objects that need to be explored in more detail (curves, holes, complex structures, etc.).

As a starting point to develop this interface, we will work with the Automated Part Inspection (API) package. API is a semi-automatic feature-based part inspector that is fully integrated with the α_1 system. This package, some of which can be seen in Fig. 6.52, enables a user with an α_1 model composed of machined features to simulate and/or drive the CMM to inspect the machined part. Using our intermediate feature-based model to guide the inspection as if it were the original, we will be able to incorporate the sense of touch into our knowledge base. With a new, more accurate model, we can loop back to the beginning of the inspection process until we have captured every aspect of the parts we inspect to the tolerances we desire.



The file bhg.cal does not exist!

FIGURE 6.52 The API user interface.

Computer Aided Design and Manufacturing

We intend to develop the CAD interface to be more accurate and to accept more complicated models. The goal is to enhance the automatic programming interface between the data obtained in the sensing module to the α_1 programming environment. The parametric and 3-D point descriptions are to be integrated to provide consistent and efficient surface descriptions for the CAD tool. For pure inspection purposes, the computer aided geometric description of parts could be used as a *driver* for guiding both the robotic manipulator and the coordinate measuring machine for exploring the object and recognizing discrepancies between the real part and the model.

The computer aided design parameters are then to be used for manufacturing the prototypes. Considerable effort has been made for automatically moving from a computer aided geometric model to a process plan for making the parts on the appropriate NC machines and then to automatically generate the appropriate machine instructions [6]. We use the Monarch VMC-45 milling machine as the manufacturing host. The α_1 system produces the NC code for manufacturing the parts.

VLSI, Uncertainty Modeling, and Languages

The software and hardware requirements of the environment are the backbone for this project. We intend to select parts of the system implementation and study the possibility of hard-wiring them. There has been considerable effort and experience in VLSI chip design [5, 8] and one of the sub-problems would be to study the need and efficiency of making customized chips in the environment. The DEDS model, as an automaton, is very suitable for path programmable logic (PPL) implementation. A number of the visual sensing algorithms could be successfully implemented in PPL, saving considerable computing time. Integrated circuits for CAGD surface manipulation is an effort that is already underway. We intend to investigate a new area: the possibility of implementing the DEDS part of the system in integrated circuitry.

Another important part to be implemented in hardware, is the "uncertainty" chip, which will provide fast decisions about the accuracy of our measurements. This is important for deciding whether the part needs more inspection steps or not. The uncertainty model depends on the nature of the part being inspected, the sensor, the strategy being used to sense the part, and the required accuracy.

There is a lot of interfacing involved in constructing the inspection and reverse-engineering environments under consideration. The use of multi-language object-based communication and control methodology between the three major components (sensing, CAD, and CAM) is essential. We intend to use a common shared database for storing data about the geometric model and the rules governing the interaction of the different phases in the reproduction and inspection paradigms [10, 17]. We have already used a graphical behavior design tool [4] for the automatic production of the sensing DEDS automata code, from a given control language description. A sensing $\rightarrow$ CAD interface has been developed as well.

6.9 Conclusions

We propose a new strategy for inspection and/or reverse engineering of machine parts and describe a framework for constructing a full environment for generic inspection and reverse engineering. The problem is divided into *sensing*, *design*, and *manufacturing* components, with the underlying software interfaces and hardware backbone. We use a recursive DEDS DRFSM framework to construct an intelligent sensing module. This project aims to develop sensing and control strategies for inspection and reverse engineering, and also to coordinate the different activities between the phases. The developed framework utilizes existing knowledge to formulate an adaptive and goal-directed strategy for exploring, inspecting, and manufacturing mechanical parts.

References

1. Aloimonos, J. and Shulman, D. *Integration of Visual Modules an Extension of the Marr Paradigm.* Academic Press, 1989.
2. α_1. α_1 User's Manual. University of Utah, 1993.

3. Benveniste, A. and Guernic, P. L. Hybrid dynamical systems theory and the signal language. *IEEE Transactions on Automatic Control,* 35, 5, 1990.
4. Bradakis, M. J. Reactive behavior design tool. Master's thesis, Computer Science Department, University of Utah, January 1992.
5. Carter, T. M., Smith, K. F., Jacobs, S. R., and Neff, R. M. Cell matrix methodologies for integrated circuit design. *Integration, The VLSI Journal,* 9, 1, 1990.
6. Drake, S. and Sela, S. A foundation for features. *Mechanical Engineering,* 111, 1, 1989.
7. Ens, J. and Lawrence, P. An investigation of methods for determining depth from focus. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 15, 2, 1993.
8. Gu, J. and Smith, K. A structured approach for VLSI circuit design. *IEEE Computer,* 22, 11, 1989.
9. Hsieh, Y. C. Reconstruction of sculptured surfaces using coordinate measuring machines. Master's thesis, Mechanical Engineering Department, University of Utah, June 1993.
10. Lindstrom, G., Maluszynski, J., and Ogi, T. Using types to interface functional and logic programming. In *1991 SIGPLAN Symposium on Principles of Programming Languages* (July 1990), Vol. 10. technical summary (submitted).
11. Marr, D. and Hildreth, E. Theory of edge detection. In *Proceedings Royal Society of London Bulletin,* 204, 301–328, 1979.
12. Nerode, A. and Remmel, J. B. A model for hybrid systems. In *Proc. Hybrid Systems Workshop, Mathematical Sciences Institute* (May 1991). Cornell University.
13. Özveren, C. M. *Analysis and Control of Discrete Event Dynamic Systems: A State Space Approach.* PhD thesis, Massachusetts Institute of Technology, August 1989.
14. Pratt, V. Direct least-squares fitting of algebraic surfaces. In *SIGGRAPH '87* (1987), 145–152.
15. Schalkoff, R. J. *Digital Image Processing and Computer Vision.* John Wiley & Sons, 1989.
16. Sobh, T. M. and Bajcsy, R. A model for observing a moving agent. In *Proceedings of the Fourth International Workshop on Intelligent Robots and Systems (IROS '91)* (November 1991). Osaka, Japan.
17. Swanson, M. and Kessler, R. Domains: efficient mechanisms for specifying mutual exclusion and disciplined data sharing in concurrent scheme. In *First U.S./Japan Workshop on Parallel* (August 1989).
18. Tsai, R. A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses. In *Radiometry - (Physics-Based Vision),* G. H. L. Wolff, S. Shafer, Ed. Jones and Bartlett, 1992.
19. van Thiel, M. T. *Feature Based Automated Part Inspection.* Ph.D. thesis, University of Utah, June 1993.
20. Willson, R. Personal communication, 1993.

Appendix A: Sample GI Joe Output

```
int State_B(VTV_ptr)
vtype *VTV_ptr;
{
     int DoneFlag;
     EventType Event;
     vtype *newVTV_ptr;
     int EventMask=0;

#ifdef VERBOSE

     printf("in state B\n");

#endif
     if (VTV_ptr == NULL) {

#ifdef VERBOSE

     fprintf(stderr, "*** ERROR: null vtv in state B\n");
#endif

        exit (4);
     };
     EventMask |= TimeOutMask;
     EventMask |= NoProbeMask;
     EventMask |= ProbeCloseMask;
     EventMask |= ProbeFarMask;
     DoneFlag 5 FALSE;
     while (!DoneFlag) {
        Event = Get_DRFSM_Event(EventMask, VTV_ptr);
        if (Event . type == TimeOut) {
            DoneFlag = TRUE;
            if (Event . fn != NULL) DoneFlag = (*(Event . fn))();
            State_ERROR(VTV_ptr);
        }
        else if (Event . type == NoProbe) {
            DoneFlag = TRUE;
            if (Event . fn != NULL) DoneFlag = (*(Event . fn))();
            State_A(VTV_ptr);
        }
        else if (Event . type == ProbeClose) {
            DoneFlag = TRUE;
            if (Event . fn != NULL) DoneFlag = (*(Event . fn))();
            State_C(VTV_ptr);
        }
        else if (Event . type == ProbeFar) {
        }
     }
}
```

Appendix B: Sample Calibration Code Output

```
Coplanar calibration (full optimization)

data file: a . pts

    f = 8.802424 [mm]
    kappa1 = 0.003570 [1/mm^2]

    Tx = −25.792328, Ty = 77.376778, Tz = 150.727371 [mm]

   Rx = −134.988935, Ry = −0.127692, Rz = −0.068045 [deg]

    R
      0.999997     0.000737     0.002416
     −0.001188    −0.706972     0.707241
      0.002229    −0.707242    −0.706968

    sx = 1.000000
    Cx = 276.849304, Cy = 252.638885 [pixels]
```

Tz/f = 17.123394

calibration error: mean = 0.331365,
standard deviation = 0.158494 [pixels]

Appendix C: Comparison Between Hough Transform and Curvature Technique

The curvature technique as implemented for this application is described in flowchart form in Fig. C.1. Using a similar analysis for a basic Hough transform implementation (just to detect circles) shows that it would require:

- *MMM* assignments (to initialize memory)
- *NMM*
  - 5 addition operations
  - 3 multiplication operations
  - 1 sqrt operations
  - 6 assignments
  - 3 comparisons
- *MMM* integers for memory

where *M* is the precision.

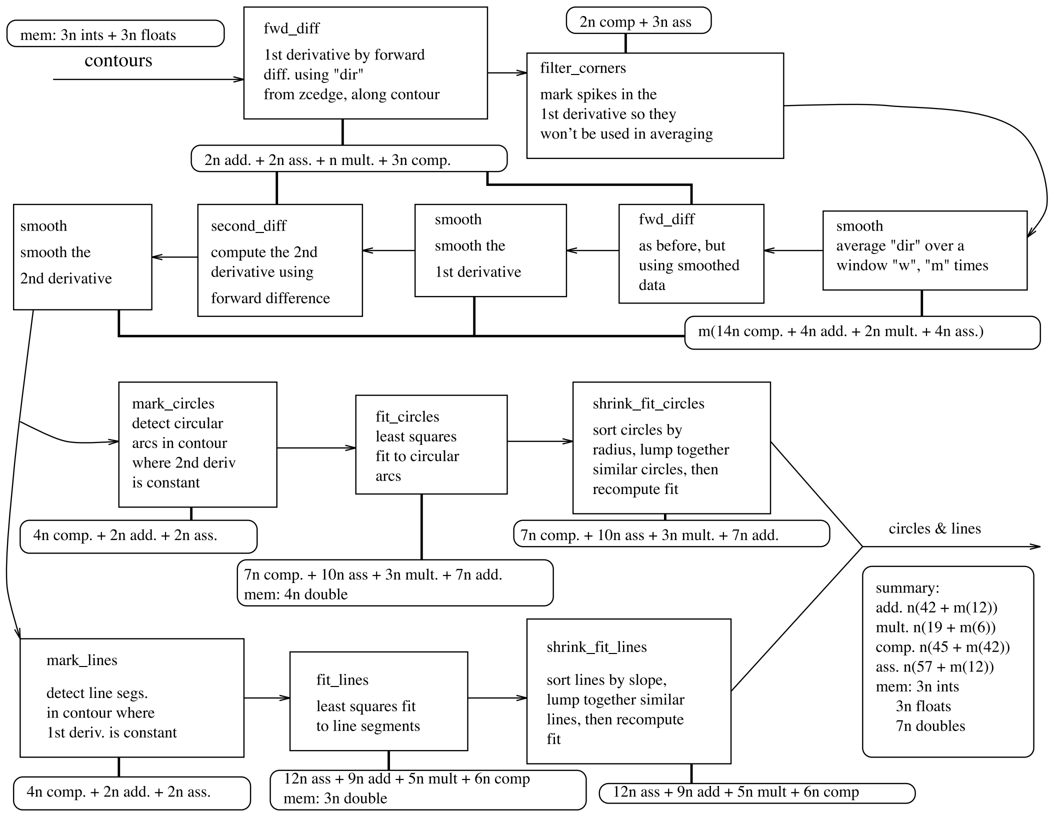

mem: 3n ints + 3n floats
contours
fwd_diff
1st derivative by forward diff. using "dir" from zcedge, along contour
2n comp + 3n ass
filter_corners
mark spikes in the 1st derivative so they won't be used in averaging
2n add. + 2n ass. + n mult. + 3n comp.
smooth
smooth the 2nd derivative
second_diff
compute the 2nd derivative using forward difference
smooth
smooth the 1st derivative
fwd_diff
as before, but using smoothed data
smooth
average "dir" over a window "w", "m" times
m(14n comp. + 4n add. + 2n mult. + 4n ass.)
mark_circles
detect circular arcs in contour where 2nd deriv is constant
fit_circles
least squares fit to circular arcs
shrink_fit_circles
sort circles by radius, lump together similar circles, then recompute fit
4n comp. + 2n add. + 2n ass.
7n comp. + 10n ass + 3n mult. + 7n add.
circles &amp; lines
7n comp. + 10n ass + 3n mult. + 7n add.
mem: 4n double
summary:
add. n(42 + m(12))
mult. n(19 + m(6))
comp. n(45 + m(42))
ass. n(57 + m(12))
mem: 3n ints
3n floats
7n doubles
mark_lines
detect line segs. in contour where 1st deriv. is constant
fit_lines
least squares fit to line segments
shrink_fit_lines
sort lines by slope, lump together similar lines, then recompute fit
4n comp. + 2n add. + 2n ass.
12n ass + 9n add + 5n mult + 6n comp
mem: 3n double
12n ass + 9n add + 5n mult + 6n comp

FIGURE C.1 Flowchart of the implemented curvature technique.

Assuming pixel accuracy, M is approximately N/π. N, for this application, can be taken to be contour length, bounded by $\sqrt{N_{lines}N_{samples}}$. Thus, the Hough tranform may be considered of order N^3 while the curvature technique used is at most order N^2. Not included in the Hough evaluation is that it would be necessary to do some sort of mode detection to determine the number of circles found.

It is anticipated that the fitting algorithm may be extended to include other conic sections than circles, and additionally that it may be extended to use three dimensional coordinates. While the Hough transform is a very useful technique, we anticipate that its memory and order requirements will grow too rapidly to meet our future needs.