Knowledge-Based 2-D Vision System Synthesis[1]

Tom Henderson and Eliot Weitz

UUCS-87-030

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

December 30, 1987

Abstract

A knowledge-based approach to computer vision provides the needed flexibility for performing recognition and inspection of objects in a complex environment. A system is described which uses knowledge about the environment, sensors, and performance requirements to construct a functional configuration of sensors and algorithms. The system uses an object-based approach to synthesize both the analytical model for an object and the executable application system.

[1]This work was supported in part by NSF Grants MCS-8221750, DCR-8506393, and DMC-8502115.

Contents

1 Introduction

Computer Aided Design systems (CAD) are becoming readily available in present day manufacturing. They have greatly increased the speed and reliability of product design and manufacturing. CAD/CAM systems have the capability to operate machinery to create designed parts without human intervention. As CAD/CAM systems become more common in the marketplace, a need for automated visual inspection becomes a logical extension to the computer aided automation process. Experimental inspection systems have been developed which use a CAD model of an object to perform visual inspection [7,19]. These systems either use the CAD representation for a part, images rendered from the CAD system, or camera images of the actual object. Systems which use information from the CAD model directly have the advantage of greater accuracy in their recognition and inspection methods. This interface between the vision system and the CAD system is a current topic of research and has brought to light the problems involved with the visual inspection and recognition of objects [14]. This paper examines the use of CAD for computer vision applications.

Automated visual inspection systems must meet certain requirements in speed, accuracy, and flexibility to be of any real use in the work environment [3]. There are several commercial 2D vision systems available using the SRI Vision Module [12]. A 2D system uses only two-dimensional features of an object from a controlled viewpoint. This immediately restricts the object's shape to be flat. These systems must operate in tightly controlled environments since they depend on global features of objects such as area, perimeter, and diameter. They are comprised of sensors, software, and processors all tuned to a particular recognition or inspection task. Generally this means they have little or no flexibility to rapidly integrate new hardware and software or reconfigure themselves for a new application. Many components of vision systems could be re-used in other applications if the system were designed with enough modularity. Ultimately it is desirable to have vision systems designed on an abstract architectural level using virtual system components which can be automatically mapped to a specific configuration. This gives the system incredible flexibility of application, and provides for a highly modularized design.

We propose here an an environment which provides a virtual architecture for the design of vision application systems. It also explores automation in the synthesis of such systems using a knowledge-based approach. This approach uses human knowledge in the form of rules to configure and operate system software and has proven a powerful method for solving complex problems [22]. A prototype vision system was designed which incorporates the use of a CAD-based model with the knowledge-based approach using information about the application, environment, hardware, and software to configure visual recognition systems. The performance of this system is also examined.

2 Knowledge-Based Systems

Any information used by a set of rules in the knowledge-based system must be represented by one or more objects. This requires that classes be well defined for the objects and the objects themselves be given well defined slots to prevent ambiguity and conflicts which rules generally cannot handle.

2.1 System Organization

2.1.1 Information Organization

As previously mentioned a rule-based system requires the creation and maintenance of a database of information for the system's operation. This database consists of objects which must be organized to best meet the needs of the vision system. The organization of this information must be robust enough to handle potential changes in the system as it evolves.

What information is needed for the knowledge-based vision system to operate? The best approach to answering this question is to acquire expertise in how vision systems operate and how they are applied to real problems, then using this expertise, determine what information is needed to solve this general class of problems. In order to design operational application systems, it is necessary to isolate how information about the world is organized by a human system designer. When designing an application system to solve a vision problem, the system designer must obtain information about: the requirement of the application, the operational environment, input of the system, and the tools and components available for construction of the system. For 2-D vision applications there are basically four independent sources from which the information for the design can be drawn. These four groups are: requirements, environment, tools, and model. Information from these four independent sources must be represented in object form and made available to every rule set in the system. Some information will be provided from outside sources such as the system user imposing certain constraints. Internal information such as information about hardware and software tools is assumed to be consistent. Outside information such as requirements and environment must be checked for consistency or a representation selected which inhibits bad combinations of data. These four groups of information can be used to define classes of objects which can then be subdivided into subclasses until the resolution of the classification is sufficient to write the rules.

There is a trade-off between the knowledge and information in a knowledge-based system just as there is a trade-off between memory and executables in a conventional system. The classification of the four major groups of information will affect the size of the rule base needed to use the information. A configuration must be selected which is generally robust for all possible applications. This is a nontrivial problem since it is hard to predict what information may be needed to configure every possible application system within the well constrained 2-D vision problem class. The organization of the underlying objects representing the information in the system must therefore be general and flexible enough to allow the addition of new information. New information added to the existing system should not require a major reclassification which could make the existing rules invalid.

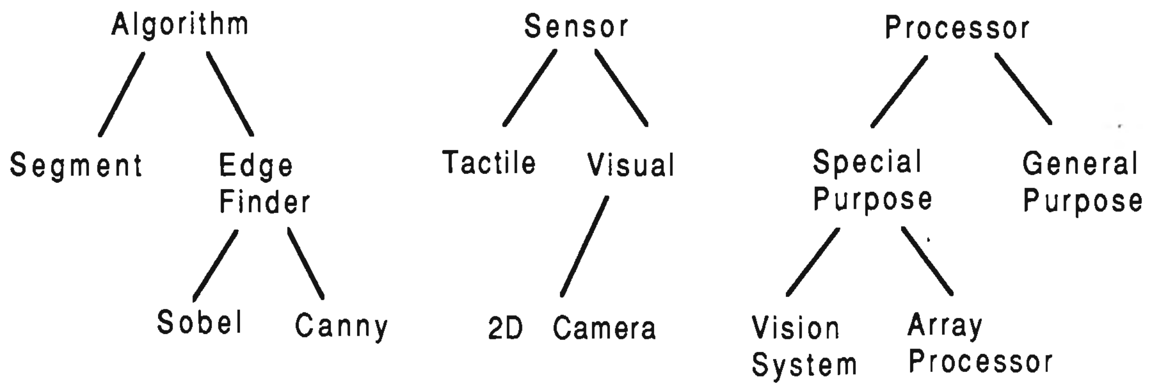


Figure 1: Information Hierarchy

A configuration was selected to classify objects within the vision system. Figure 1 shows the tree like structure of the classes and subclasses for the different components. Subclasses inherit the characteristics of their parent classes, characteristics being their slots and methods. This organization provides a logical and robust definition for all objects within the system. Subclasses may be added to the tree with minimal affect to existing rules and objects (depending on the level in which they are added).

2.1.2 Knowledge Organization

Organization of the knowledge encoded as rule sets within the knowledge-based vision system is perhaps the most difficult part of the system design. Problems must be isolated within the vision system for which rules can be used in their solution. As mentioned earlier, these problems will be higher level tasks which humans usually perform. The rule sets for the vision system, therefore, contain high level rules which depend on conventional programs to solve lower level implementation problems. Each of these system subproblems must have well defined domains for their respective rule sets. The size of the problem domain will generally dictate the number and sophistication of the rules required to solve it. Initially domains are kept well constrained in the knowledge-based vision system with the intention of future expansion.

Four general sources of information were defined previously which provide sufficient information to synthesize an application for a given problem. The knowledge needed to configure an application system from these sources can be divided into several subproblems each with a domain suitable to be solved by a set of rules. Each subproblem corresponds to a design "step" taken by a human application system designer. The overall process can be grouped into the following design steps:

1. Selection of an application whose functionality and performance matches that of the required system.
2. Adaptation of the application to the criteria of the new application.
3. Verification of the new system's performance in the given situation.

These three steps are high level descriptions of what is required by a system designer to perform the task. Information required for each problem domain overlaps (i.e., to both select, adapt, and verify an application requires the same knowledge of the environment) so it must be made available to each rule set and each rule set is affected by changes in this information.

The system knowledge corresponding to the first step is called the *application selection rules.* These rules use information about the requirements, environment, sensors and algorithms to determine which applications in the knowledge base can be applied. The second step is performed by the *application specific rules* which are subdivided into sets according to applications. Each application rule set uses information in the database to specify an application system within a given framework of its particular application. The knowledge to solve the last subproblem is captured by a group of rules called the *application verification rules.* These rules embody the knowledge needed to analyze results of a simulated application system to determine if any knowledge in the system requires modification in the form of new rules or the elimination of existing ones. This provides a learning capability to the knowledge base and is discussed in Section 6. A graphical depiction of the knowledge-based system can be seen in Figure 2.

Application Selection Rules The application selection rules' job is to use information about the requirements, environment and available applications to select a suitable application. The domain is restricted by the amount of information available for the selection of an application. The designer of the rules determines what information from the requirements and environment will be used to make the selection decision. This set of rules should remain flexible during the lifetime of the knowledge-based vision system since the knowledge contained in the rules can only be obtained from implementation experience.

The main advantage in using rules to select an application is that several applications can be selected and synthesized in parallel. The human designer would most likely be selective and prioritize the applications selected to save wasted effort on applications which do not appear promising. The application selection rules do not have to be as selective and may select as many applications as the system will bear. An increase in the number of applications available may of course require that the application selection rules discriminate more in their selection of applications.

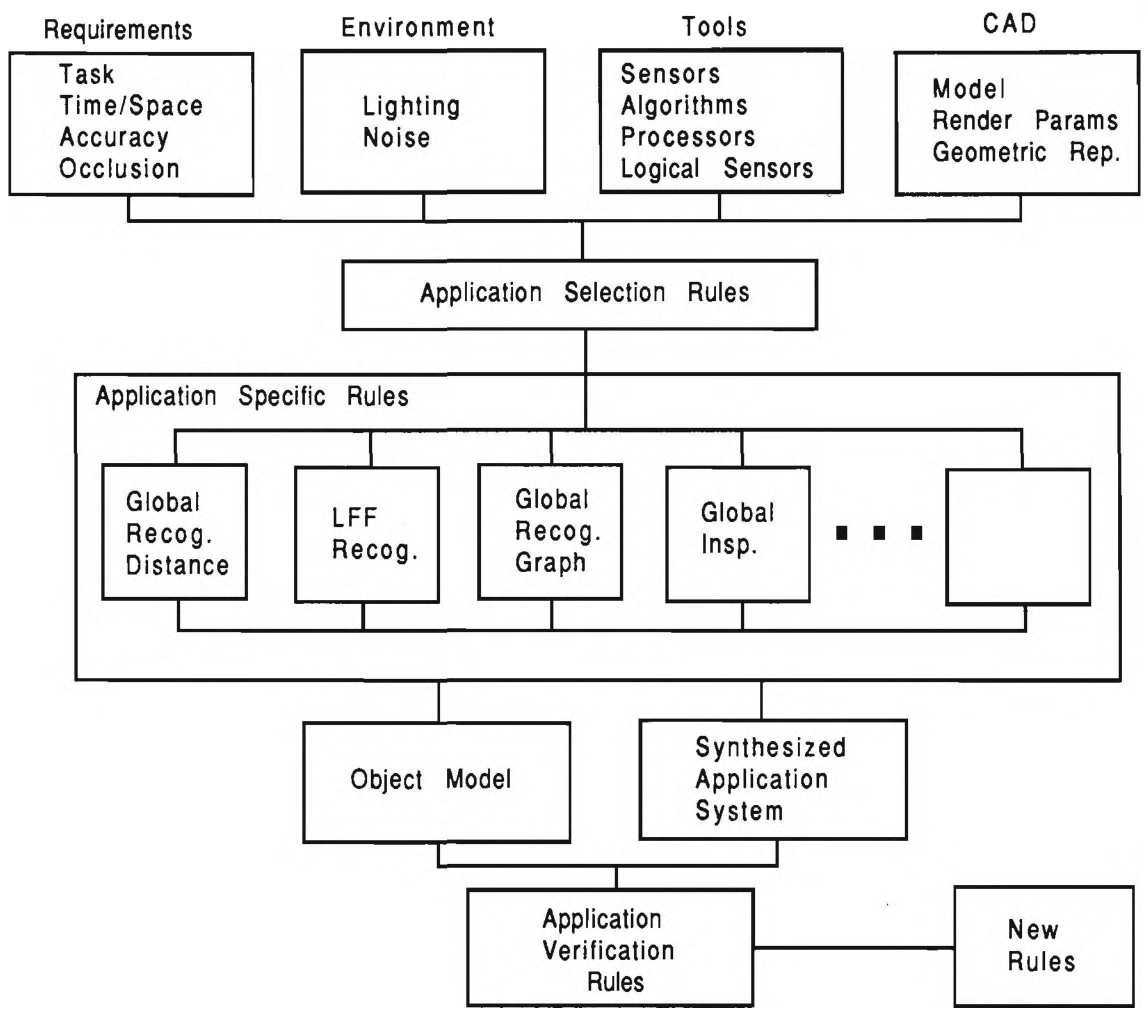


Figure 2: General Components of a Knowledge-Based Vision System

Application Specific Rules The application specific rules are a collection of subsets rather than a single set of rules. Each of these subsets has the job of adapting its particular application to the situation posed. The rules require specific information regarding their application's components and performance in addition to the same information used by the application selection rules. Most of this information is available in the slots of objects representing the tools. The output from these rule sets is the application itself, that is an executable image of the application along with any data it might need such as a CAD model. When interfacing a rule set with a set of conventional programs, there is a balance between decisions made by the rules and those made by the programs. The rigidity of the program structure determines the flexibility of the problem domain. A very rigid conventional system design would not need rules to operate. The same system can be parameterized to give some flexibility and a need for rules. The applications in the vision system can be described as parameterized fixed systems. Their general control structure has been defined and the rules determine what components fit inside of the structure as well as define through parameters how each component operates. This restricts the problem domain sufficiently while providing enough flexibility for creating application systems of varying functionality.

Application Verification Rules The application verification rule set requires the most sophistication of the three rule sets. This is because they must interpret output from applications and correlate this information with the design criteria. Their output is in the form of new rules which may replace or supplement existing rules in both the application selection and application specific rule sets. These rules may need to be divided into subsets according to the applications being verified. They are not as well defined as the other two rule sets since they are not essential to the vision system's operation and have been defined more for future vision system expansion.

2.2 Application Implementation Considerations

Using the high level design just described for the knowledge-based system, certain details on how the applications within the system are to be implemented must be defined. Each application will require the custom integration of hardware and software components; therefore it is important to provide an environment which is both powerful and flexible for application implementation. Since the knowledge-based vision system uses the object-based approach in its design, it makes sense to extend this concept to the implementation of the application itself. This means that objects are used to represent both hardware and software components in the system. The hierarchy of object classes which define each component for use in the system provides the means for representing each system component through a class instance. These instances not only define the necessary information about the component for use by any rules but also have the information needed to execute or operate the component. Instances of the same object class make it possible to represent a piece of hardware or software in two separate applications without duplicating the component itself. Most vision applications share many of the same components; therefore classes can be shared between them. This is consistent with the idea of synthesizing new application systems from old system components.

2.2.1 Multisensor Integration With Logical Sensors

Logical Sensor System Specifications address the problem of interfacing hardware and software components within a multisensor system. It defines a standard building block for the design of multisensor systems [13]. Figure 3 gives a graphical representation of the basic sensor. The output of each logical sensor is specified by the ***characteristic output vector.*** Commands are accepted through the sensor's ***control command interpreter.*** These Logical Sensors can be configured as a hierarchical network where control information flows downward and sensor output upward through the hierarchy.

This method of system design provides a coherent description of multisensor information flow. The internal design of the sensor allows for alternative subnetworks of sensors to be used for dynamic reconfiguration in response to sensor faults or environmental changes. Logical Sensors are a powerful tool for configuring multisensor systems in an object-based environment. They can be viewed as building blocks for a virtual multisensor architecture in which complex systems requiring dynamic allocation of resources can be specified. Logical sensors specifically address such issues as fault tolerance and distributed processing in a multisensor system as well.

Using logical sensors to configure an application system requires that the logical sensors be mapped to actual components in the system. This can be done by assigning a logical sensor to each component or grouping the function of several components into a single logical sensor. The logical sensor would then provide for the running and interfacing of its internal components to other sensors and the outside world. It can be readily seen that designing complex logical sensors with many internal components is a hard task for humans let alone application specific rules. It therefore seems more practical to use logical sensors that have been prefabricated containing one or more components to provide functions which are commonly performed in the application.

2.2.2 Application Synthesis

The application specific rules have the responsibility of configuring application systems within the constraints of the requirements and environment. Configuration of the application means to have all of the components needed for the operation of the application ready for execution. Generally this requires that instances be created for each component of the application and configured to operate in the environment. Parameters for the operation of the components must also be determined.

For model-based recognition applications the rules are required to: define the components and parameters for the training system, train on the CAD model, produce parameters for the recognition system to operate, select components for the recognition system, and deliver as a final result the model plus an executable instance of the recognition system. The amount of work done to accomplish this task depends on the flexibility of the control structures provided for the training and recognition systems. Initial applications will have well defined procedures for both systems reducing the size of the overall task. The amount of flexibility an application has for meeting its requirements is determined by the flexibility of the control structures provided for training and recognition. As knowledge is gained about the implementation of the application, the control structure may change and become more flexible to match the increase in knowledge represented in

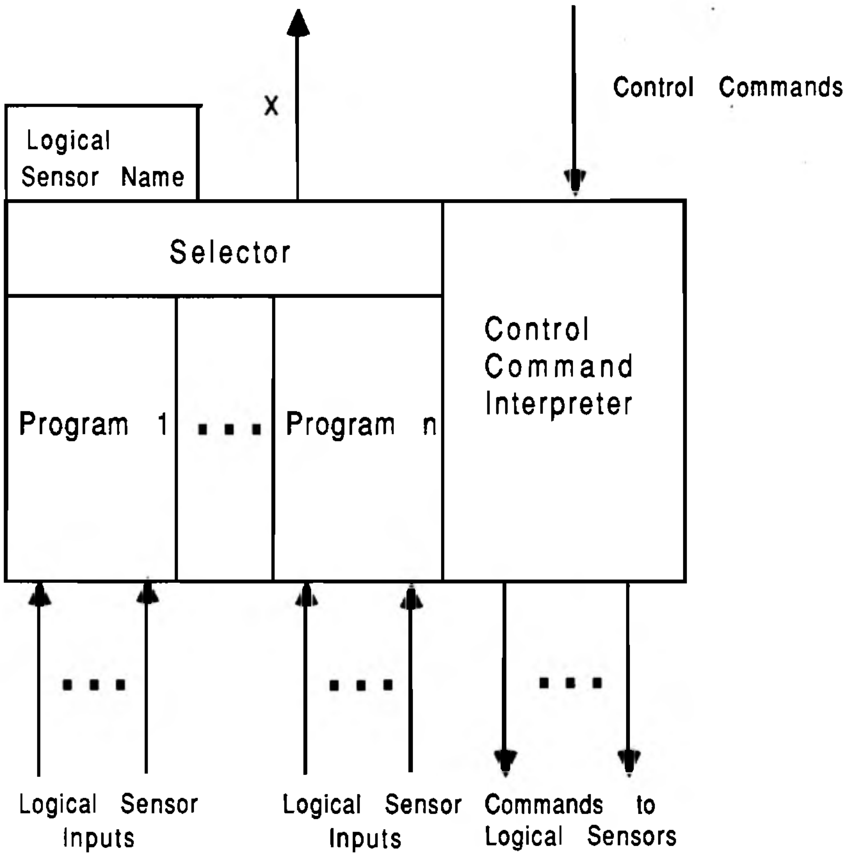


Figure 3: A Logical Sensor

the rules. The size of the requirements and environment problem domain is eventually restricted by this flexibility (i.e., heavily restricted applications do not have the flexibility to match a wide range of requirements).

3 Visual Recognition Methods

Visual recognition by machine is the most common problem tackled by industrial vision systems [16]. The problem usually entails a conveyor belt or bin containing several parts in random order and orientation from which a part must be selected, identified and localized, information which would most likely be used to manipulate the part. The complexity of the general case for the bin picking problem is large since a part may be arbitrarily oriented and be partially occluded by another part. To reduce the complexity of the problem, most industrial vision systems rely on mechanical mechanisms that force parts to lie in a well constrained orientation.

For the purpose of this paper we will adopt several restrictions to the environment and how the objects in a scene may be oriented. One restriction is to not allow a part to be seen in more than one or two stable positions. This means the system is restricted to use flat parts with few or no three-dimensional features. Objects may occlude one another but may not occlude themselves as may be the case with flexible material. This environmental restriction does not apply to the overall application potential of the vision system but rather to the applications implemented for the prototype system.

Model-Based object recognition describes a class of 2-D object recognition methods with the same overall recognition strategy. They use features extracted from an object to build a model representing that object. The features represent characteristics of the shape, size, or color of the object which make it unique. Usually more abstract models require more sophisticated feature extraction techniques. The features may range in complexity from simple pixel counts to a polygonal estimation of the object.

Construction of this model is usually called the *training* phase of model-based recognition where the system becomes "acquainted" with the object. The second phase of the recognition method is called the *classification* phase. This involves matching a model of an object to an object in an actual scene. Both phases should use the same feature extraction techniques for consistency but this is not a requirement.

Previous vision systems have a training mode that allows the user to put selected objects in front of the imaging system to have their features extracted and recorded. Since the knowledge-based vision system discussed here interfaces with a CAD system we can extract all needed features from the CAD model itself. This provides for more accurate models since features are directly extracted from an ideal part. Many methods are used for 2-D model-based object recognition. The global feature method and the relational graph method [9] have proven themselves and are used here.

3.1 The Global Feature Method

The global features method uses global features of an object such as its area, perimeter, diameter, and center of mass. The object can be characterized by creating a feature vector consisting of n feature values. The uniqueness of the vector in feature vector space depends on the number of feature values and the robustness of each. Representing the object as a feature vector has the advantage of using vector arithmetic to evaluate and compare objects. A Euclidean distance function can be used to determine the distance between an object and a model.

This method also has the advantage of being fast and simple in its feature extraction and matching techniques, but not without certain weaknesses. One problem is that distances in feature space do not correspond uniquely to differences between images of objects. This means that offsetting differences of two features in the object and model can cause erroneous matchings. Differences in feature values having higher magnitudes may cause a greater fluctuation in the distance value than would differences in features having smaller magnitude values. This requires that feature values be normalized to solve this problem.

Normalizing the feature values gives each feature equal weight in calculating the distance value. It would be desirable to weight certain features higher than others to give robust features more decision power when calculating feature vector distances. Using the model object rotated in several positions, rotational variances can be measured for each feature and robustness determined as a result. Features with low rotational variation are generally more robust than features of high rotational variation. The variance itself is normalized and used as a weight in the distance function.

Generally, global features are invariant to rotation and translation yet quantization error can cause sizable variance in feature values as a result of rotation. Certain global features such as area and perimeter are effected significantly by noise since pixel counts will not be accurate. Global features completely fail for an occluded object since they are statistical values based on the entire object being visible. This problem can't be corrected and global vision systems must have a mechanism for preventing occlusion. In general, global features are simple to calculate but are costly to compute since every pixel in the image must be considered in the calculation. Object models based on only a few global features can be efficient in overall cost but are less robust.

3.2 The Relational Graph Method

The relational graph method takes the approach of extracting what are called local features and using relations between them to model an object. Local features are generally cheaper to compute since each pixel in the object image need not be examined for feature calculation. In some cases they can be extracted directly from a CAD model rather than from gray level images.

These features are grouped into specific classes according to their values. Feature values may be corner angles or hole areas, etc. Both these values and relative values are used to build the relational graph model.

Object classification is performed by extracting local features from an image, classifying these features according to the same conventions of the model, and building a graph whose nodes represent model and image feature pairs in the same class.

Let $Node_1$ be (l_1, m_1) and $Node_2$ be (l_2, m_2). An arc may be drawn between $Node_1$ and $Node_2$ iff:

1. The Model features are not the same (i.e., $m_1 \neq m_2$).

2. The Image features are not the same (i.e., $l_1 \neq l_2$).

3. The relational values of the image features are similar to those of the model features (i.e., $value(m_1) \cong value(l_1)$).

Complete subgraphs represent a group of mutually consistent nodes, called a clique, which represents a possible match between model and object. Figure 4 shows a set of model features and image features which have been organized into graph form. We can see from the graph that there is a complete subgraph made of nodes {1,2,5,6}. These nodes have mutual consistency and will be treated as a hypothesis. If more than one clique exists in the graph, each is treated as a hypothesis which must then be verified. Hypothesis verification usually requires that additional information is extracted from the image to determine the correctness of a hypothesis.

There are several advantages to this method of object recognition. It is capable of matching a partially occluded object to a model since only a subset of the object's local features are needed to form a hypothesis. There are weaknesses in the method however, more specifically in how the model is built and subsequent graph matching performed. Graph matching is an NP-complete problem so great care must be taken in what features and relations between them are used to build the model [1]. A large graph can create a large search space which can make the method quite slow. The Local Feature Focus relational graph method suggests the use of local feature clusters to keep the graph search space for object classification to a minimum. There are many advantages to this method; it requires that local features be extracted only once and clusters sized according to speed and accuracy requirements. It also requires that the objects must have a specific type of complexity in their shape such as corners and holes. For simple objects or nonpolygonal type objects, the method may fail completely since no holes or corners exist.

3.3 Local Feature Focus

Local Feature Focus is a relational graph method developed by Bolles and Cain [6]. Two types of local features are used: holes and corners. The model is built by first extracting holes and corners from the object and selecting values for classifying them. Feature classes are defined by setting a tolerance for the feature value variance. This tolerance is dependent on the sensitivity of the feature extraction algorithms as well as the feature value variances of the model itself. Figure 5 shows a sample object with class and local features extracted. The sample object has features with values which are distinct enough that the tolerance for selecting the classes is independent of the object. If the object had all of its corners within a 10 degree range and the tolerance is greater than 10 degrees, there will be only one class for all of the corners which will reduce the robustness of the model.

Focus features are selected from the extracted class features. Several or all of the classes are ordered by their robustness as a class feature. This information can either be extracted through a symmetry analysis of the object or through knowledge gained from previous systems built around a similar object. The focus features are used to form clusters which consist of a number of secondary features, their distance and relative orientations to the focus features. The number of focus features and the cluster size for each determines the size of the search space for model matching. Table 1 shows the focus features and their respective clusters for the sample object. A polygonal approxi-

$$features_{model} = \{mf_1, mf_2, mf_3, ...mf_m\}$$

$$features_{image} = \{if_1, if_2, if_3, ...if_n\}$$

Model/Image Feature Pairings

node 1: (m1,f3)	node 4: (m1,f7)
node 2: (m2,f4)	node 5: (m3,f5)
node 3: (m4,f2)	node 6: (m4,f6)

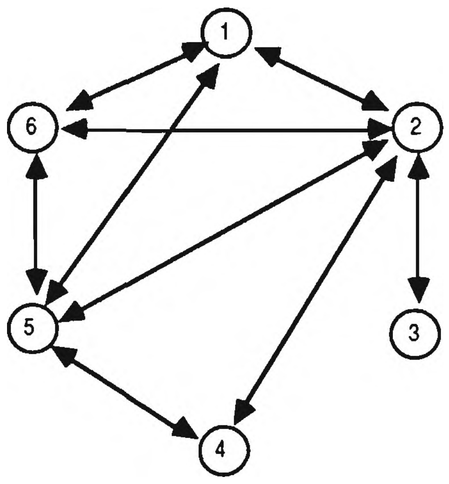
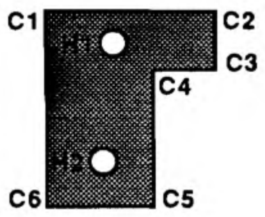


Figure 4: Relational Graph Example



Local Feature Classes	
H: Type: Hole Area: 30	B: Type: Corner Angle: 270.0
A: Type: Corner Angle: 90.0	

Local Feature List

Lbl	Type	Location	Orientation
H1	H	(0.39,-0.20)	
H2	H	(0.31,-0.75)	
C1	A	(0.00, 0.00)	-45.0
C2	A	(1.00, 0.00)	-135.0
C3	A	(1.00,-0.27)	135.0
C4	B	(0.55,-0.27)	135.0
C5	A	(0.55,-1.00)	135.0
C6	A	(0.00,-1.00)	45.0

Figure 5: Feature Extraction From an Object

mation of the object is needed to complete the model of the object. This is used as a template for hypothesis verification which is described later.

The local feature focus method uses this idea of focus features and clusters to reduce the search space when matching an object to a model. The cluster size restricts the number of nodes in the graph and provides a mechanism for adjusting to speed and accuracy requirements. In the Bolles and Cain technical report [5], the following recognition strategy is described:

1. Locate a focus feature.
2. Locate its nearby co-features.
3. Identify a cluster of image features.
4. Construct a hypothesis regarding the identity, position, and orientation of an object.

Table 1: Focus Features and Their Clusters for the Sample Object

Focus Feature	Secondary Feature	Distance	Orientation
H1	C1	0.20	-
	C3	0.61	-
	C4	0.16	-
H2	C5	0.35	-
	C6	0.39	-
	C4	0.54	-
C4	H1	0.16	-
	C2	0.52	-270.0
	C3	0.45	0.0

Each maximal clique in the graph of consistent features represents a hypothesis which must be verified. This reduces the possibility of false matches. The system verifies hypotheses by using the rotated and translated polygonal approximation from the model as a template which is overlaid onto the object. The rotational and translational offsets can be obtained by using the relative rotation offsets between model and object features. Verification consists of a series of dark-to-light transitions measured perpendicularly at regular intervals along each edge of the template. Weights are assigned to favorable and nonfavorable transitions the sum of which is used to accept or reject the hypothesis. Figure 6 shows graphically how the measurements are taken with the template.

If the hypothesis is rejected, another is selected and verified until there are none remaining or one has been positivly verified. If the result is positive, the model object is reported as found with its position and orientation in the image. The system removes the features of verified hypotheses from the graph and continues creating and verifying new hypotheses until all cliques have been exhausted.

Local feature focus is a fast and reliable way to locate 2-D objects within a scene and can operate in a noisy environment as well as recognize partially occluded objects. It can be adapted to work with different types of features and can be adjusted to work with varying accuracies and speeds. This combination of speed and robustness makes it a superior 2-D recognition technique.

Figure 7 shows the control structure for the Local Feature Focus application. It outlines the procedure used to generate the application model and executable system. To meet time, accuracy and space requirements different segmenters or feature extractors may be defined for use within the procedure as well as the parameters and tolerances specified by the system specifications which are determined by the application specific rules.

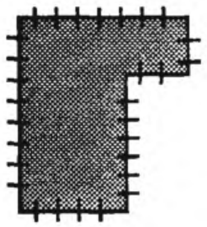


Figure 6: LFF Hypothesis Verification

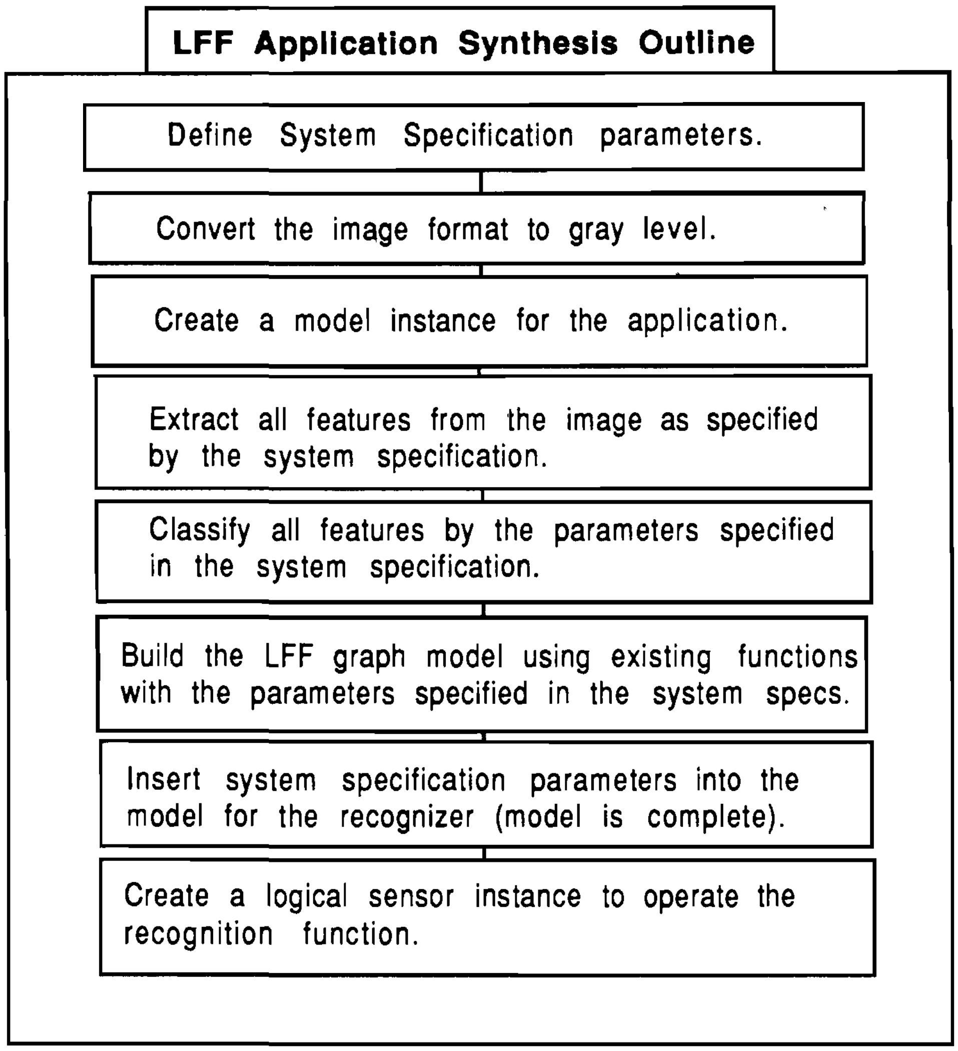

LFF Application Synthesis Outline
Define System Specification parameters.
Convert the image format to gray level.
Create a model instance for the application.
Extract all features from the image as specified by the system specification.
Classify all features by the parameters specified in the system specification.
Build the LFF graph model using existing functions with the parameters specified in the system specs.
Insert system specification parameters into the model for the recognizer (model is complete).
Create a logical sensor instance to operate the recognition function.

Figure 7: Control Structure for the LFF Application System

4 System Implementation Considerations

With the basic framework for a knowledge-based vision system defined, the next step is to select a knowledge base representation language and environment. Object-based programming is best implemented in a functional programming language environment. Lisp provides substantial object support and has proven a good environment for AI applications. Rule systems such as OPS5, MRS, and HPRL have been implemented in the Lisp environment and expert systems shells such as EMYCIN have also been based on Lisp [8,17]. The Lisp environment seems to be a good choice for knowledge based system development.

4.1 FROBS

FROBS (FRames + OBjectS) is a Lisp package which supports both frames and objects [21]. It is a "best of both worlds" application, boasting the speed of Common Lisp objects and the benefits of frame world. FROBS is written in HP Common Lisp and also runs in PCLS [23]. FROBS come in two basic flavors, class FROBS and instance FROBS. Class FROBS serve as the templates from which multiple objects can be instantiated. They can inherit characteristics from one or more "parent" FROB classes. Figure 8 shows how an algorithm class FROB and a subclass FROB are defined.

The basic building block of the FROBS package is called a module. Modules consist of a class FROB and all of its associated methods. This provides for total method and data access hiding with no distinction between methods and slots. The organization of class FROBs can be viewed as a tree structure, although more complicated schema-type structures are possible through multiple inheritance. FROB class instances are leaves of the tree. Figure 9 shows how FROB classes can be organized into a hierarchical structure with instances as leaves.

FROBS are used to build both the knowledge-based vision system and the application system it synthesizes. It seems only logical to have an object-based application be the byproduct of an object-based knowledge system. This allows templates in the knowledge-based system to be directly used in the application system itself rather than performing a transformation to a some other target language. The concept of logical sensors can be implemented easily using objects to form logical sensors. Class FROBS can represent logical sensor templates to be instantiated for application system synthesis.

Most importantly the FROBS package provides forward chaining rules as well as slot daemons. Slot daemons can be useful for automatic data consistency checking and hidden slot calculations. The forward chaining rules provide the mechanism needed to create the knowledge base. Figure 10 shows the syntax of a FROB rule. Overall The FROBS package provides excellent support for object-based programming and is used to design the knowledge-based vision system.

```
(def-class algorithm nil
           :slots (name
                   size
                   language
                   machine
                   executable
                   args
                   input
                   input-type
                   output
                   output-type
                   e-function
                   machine-out
                   complexity
                   stability
                   robustness
                   p-factor))

(def-class feature-calculator ({class algorithm})
           :slots (feature-type
                  focus-type))
```

Figure 8: Example of FROB Class Definition

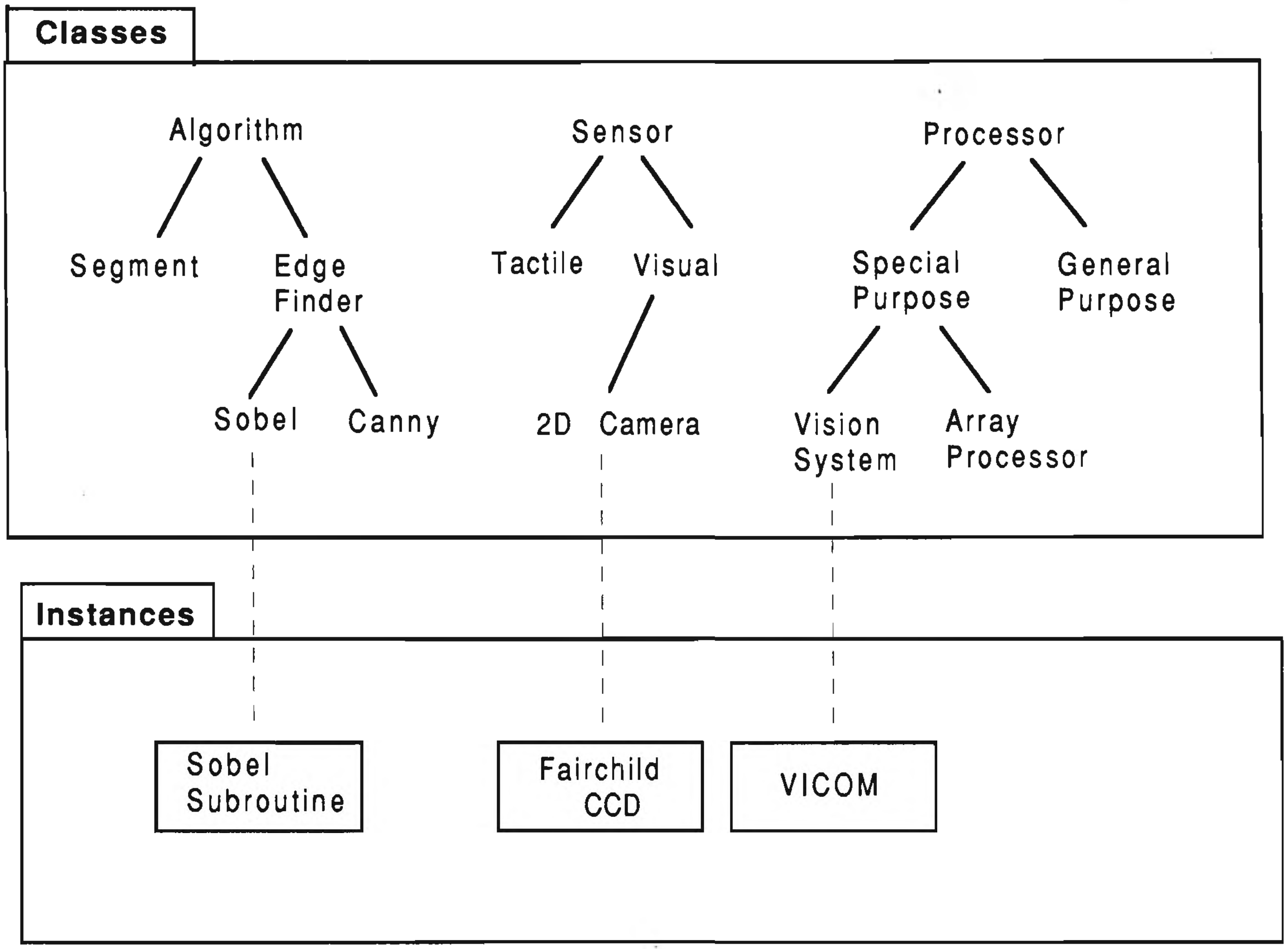


Figure 9: FROB Hierarchy

```
(def-rule select-lff
  :type ((?req requirements))
  :prem ((not (member 'lff (applications ?req)))
         (equal 'recognition (task ?req)))
  :conc ((assert-val ?req 'applications
                     (cons 'lff (applications ?req)))
 (make {class system-specs}
       :task 'recognizer
       :method 'lff
       :time (time ?req)
       :space (space ?req)
       :accuracy (accuracy ?req))))
```

Figure 10: A FROB Forward Chaining Rule

4.2 System Support

The knowledge-based system must have utilities for supporting the networking of logical sensors and objects. These utilities provide the foundation from which the system is built. Higher level utilities are built on top of lower level ones for sophisticated system operations. The lowest level utility functions should have a maximal amount of flexibility since it is not known what or how more powerful constructs built upon them will be used. In the prototype system they are implemented as methods attached to FROB classes which define major components of the system. These classes and their methods form the templates from which application systems are synthesized. The application specific rules which use knowledge of these templates to construct a system rely on the transparent nature of the methods to handle lower level hardware or operating system specific tasks. An example is a FROB representing a class of cameras. Knowledge known about operating this class of camera is represented in a "run" method which is local to the class. It executes operating system commands which are not of concern to the object using the method. A "run" method would also be provided to other sensors which have other operating system commands which are transparent to the caller of the method.

4.2.1 Language Incompatibilities

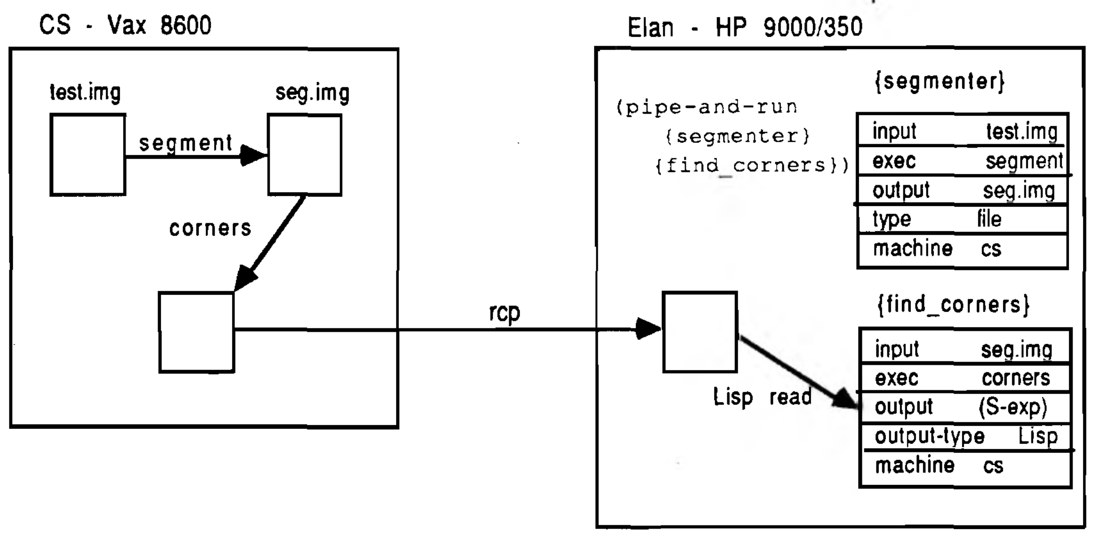
Since the application system is created from FROBS in the same environment as the knowledge-based system, the application system runs in the Common Lisp environment. To require that all of the algorithms in the system be written in Common Lisp would be a severe restriction to its flexibility. The object-based approach allows algorithms written in any language to be incorporated into the system as an algorithm object. Methods are used to run the algorithm and provide it with the necessary I/O. Since the internal representation of the object is transparent to I/O from the outside, algorithms written in any language can be incorporated into the system as long as there are low level utilities in the system to support the methods which run them.

4.2.2 Object Communication Protocol

When designing a multisensor vision system using objects, there must be a well defined way for one object sensor to pass information to another. Logical sensors address this problem in an abstract sense, but a specific protocol must be decided which has the flexibility to accept all kinds of data. The protocol is represented in the slots and methods of the logical sensor objects. There must be a way to pass information from machine to machine as well as an efficient way to pass information in the Lisp environment itself. We can separate the two as different types of information passing, file piping and S-expression passing.

Passing S-expressions between objects is a trivial task. All that is required is a slot in the algorithm object which stores the expression to be passed. This slot can be read by any object requiring the expression as input. To perform file piping on any host in the system, the simplest approach is to use the Unix pipe facility which allows executables to work as filters passing their output to the next program in the pipe. This is the easiest implementation since it is supported by the remote shell command "rsh" which is used to perform tasks on remote machines. It requires, however, that most programs written for the knowledge-based system be written in filter form on machine supporting Unix. This is not an unreasonable requirement since Unix is a widely supported operating system and it is good modular style to have a system designed with filters. Other programs can be run as well as filters although it is up to the user to supply names and flags in slots of the object which contains the program. Only filters can be handled "automatically" by the piping method.

At some point in the multisensor system's operation, information from a remote machine will have to be read by an object in the Lisp environment or vice-versa. This requires some special processing on the part of the methods performing the pipe. To send the S-expression output of a Lisp algorithm to a non-Lisp algorithm, certain conventions must be adopted. The program receiving the S-expression must know that its input is in such a form. Each algorithm in the knowledge base must have information regarding what format it expects its input to be in and what format is produced as output. This can be done with methods using slot information in the algorithm object. These methods determine what format conversions are necessary for information piped between algorithms. Information transfer between machines is performed when objects have slots indicating that their executables are on different machines. Figure 11 shows how a pair of objects in the multisensor application system uses their methods and slot values to pipe input.



"segment" and "corners" are filters which run on the CS Vax. They are executed remotely by the "pipe-and-run" method on Elan.

{segmenter} and {find_corners} are algorithm class object instances which represent filters on CS Vax. The "pipe-and-run" method invokes the filters and passes output from one to the other. The result of the "corners" filter is copied to Elan and read as an S-expression as specified.

Figure 11: Piping of Information Between Objects

5 System Overview

Application software systems are usually designed around restrictions created by hardware and supporting software. The knowledge-based vision system helps eliminate some of these restrictions by hiding the underlying system from the applications with methods that invoke utilities to deal with the underlying system. A prototype configuration for the vision system was created with methods to hide the lower level system functions. The applications still had to be aware of the capabilities of each of the processors to be efficient in their use. The allocation of this resource for the system configuration was determined by the author and might require change if ported to new hardware. This is only to be expected as it is beyond the scope of this work to configure systems with knowledge about how to best utilize hardware and system software resources, but it may be a consideration for the expansion of the knowledge-based vision system.

The hardware available to implement the prototype vision system included an HP 9000/350 series "Bobcat" running HPUX (System V Unix variety) and HP Common Lisp, a Vicom image processing system, a Fairchild CCD camera, and several vaxes running 4.3 BSD Unix, all connected via an ethernet interface. Every processor in the system configuration with exception of the Vicom can access files using a remote file server; however this proved unreliable in the early prototype system and was replaced by standard Unix network utilities. Both Unix and the ethernet made the implementation of the inter-object protocol simpler than it could have been with other operating systems. A diagram of the hardware configuration for the vision system is shown in Figure 12.

The vision code was placed on the machine closest to the image acquisition hardware; the SP Vax. SP is a significantly weaker Vax than CS computation-wise so CS was used for computation even though the executables were stored on SP. This was possible due to their similar architecture which permitted executables from one to be run on the other. The other Vax, GR, is the home of Alpha-1 and is directly connected to the NC milling machine. The only hardware in the system supporting FROBS was the Bobcat (HP 9000) which also was connected via the ethernet. Executables that could be shared across the Vaxes could not be run on the Bobcat, so it was isolated as far as sharing executables with the rest of the hardware. The Bobcat was used as the home for the high level functions of the vision system. Its programming environment contained the objects in the vision system and it acted as a central controller for the vision system. Algorithm objects and logical sensors located on the Bobcat which represented code written on other machines used system utilities on the Bobcat to run remote shells over the ethernet. All inter-machine object communications were done by copying temporary files across the ethernet as well. High level Unix utilities were used to do this such as "rcp," which allows for easy implementation as well as the immediate integration of any processor with a dialect of Unix into the vision system.

This hardware and software configuration was used in combination with the vision system support utilities just described to create and test the prototype vision system and its applications, the results of which will be discussed in this section.

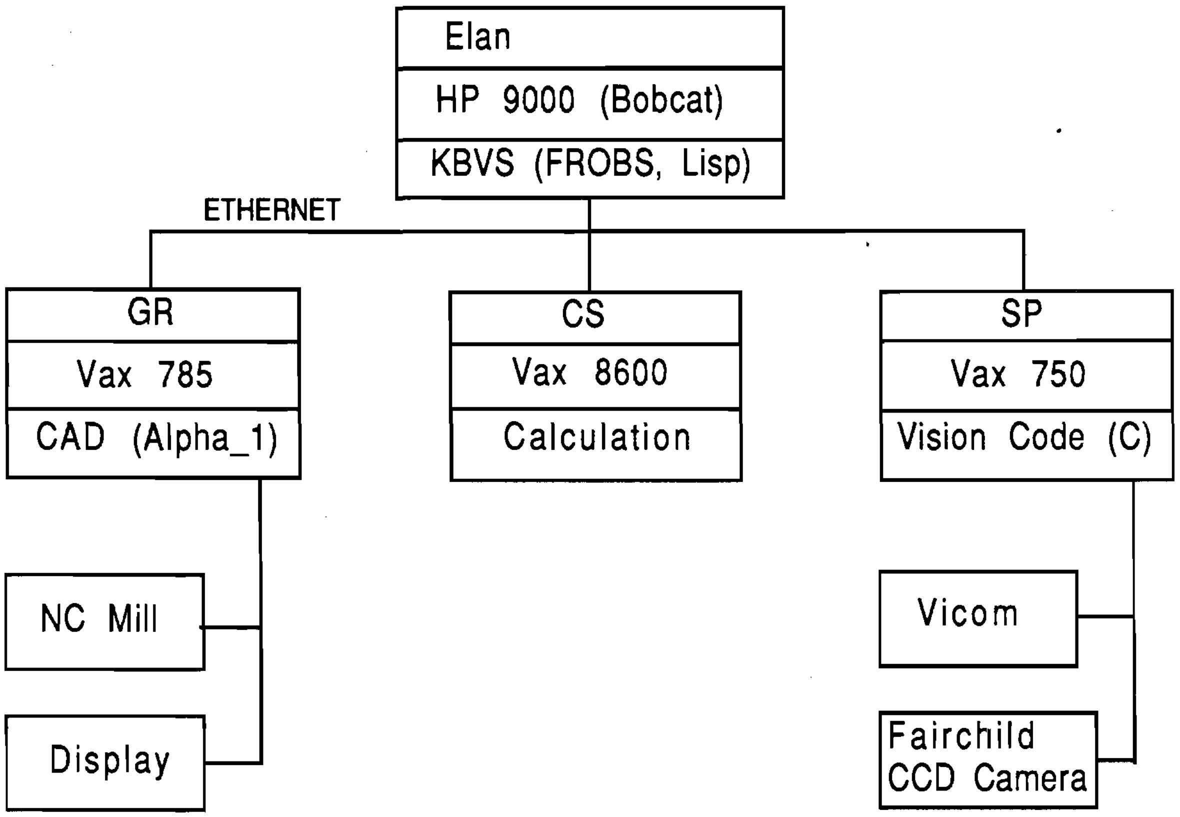


Figure 12: Knowledge-Based System Hardware Configuration

5.1 Rules and Requirements

The first prototype system designed was mostly hardwired to use the global recognition application system exclusively. For this early system, rules and requirements were not defined. As the system grew more sophisticated with the implementation of the Local Feature Focus application, requirements and rules were used to configure the applications as previously described. A "requirements" class FROB was defined to create a template for organizing the requirements information. Slots in this FROB were assigned values by the user of the vision system to define the task requirements. The values were compared relatively by the application selection rules which used these values in their premises to begin the application synthesis process. Figure 13 shows the definition for the requirements FROB class along with a defined instance used to test the vision system. It can be seen that the "time," "space," and "accuracy" slots are ordered by decreasing value. This means that time is more important than space which is more important than accuracy. The magnitude of the differences between these slot values can be used to convey to the vision system the importance of each to the user. The "applications" slot is a multiple value slot which contains the applications selected by either the user or the application selection rules. The user may specify what applications are applicable to the task required or let the application specific rules decide. In the example both prototype system recognition methods were selected by the application selection rules since they both can be applied due to the lack of occlusion.

The output of the application selection rules is one or more instance(s) of a "system-specs" FROB class. Each instance represents an application method which the selection rules deem appropriate for the requirements. The slot values of the system-specs instance represent parameters which determine how the model and application system will be created during training. Figure 14 shows the system specifications defined for the requirements in the previous figure. These values are defined by both the application selection rules and the application specific rules which begin the training process. When the necessary parameters for the application system have been defined, the training process begins. Training is performed by a Lisp function which can be done manually, as is for system testing, or by an application specific rule. The trainer uses the parameters defined to perform its task and produce as a result two FROB instances, one a model class instance FROB containing model information and the other a logical sensor instance representing the application system. The system specification instance seen in Figure 14 was created with parameters to match the high speed, low accuracy values defined in the requirements.

The number of application specific rules needed for an application depends on the constraints of the application implementation. For the global feature application there theoretically exists more flexibility in what features can be used as well as how the model can be configured. In practice, however, the overwhelming robustness of the Area feature made the recognition application of this method inflexible. Local feature focus, on the other hand, was relatively well constrained with flexibility only in its tolerances for prototyping, clustering and recognizing local features. In practice these parameters proved highly useful in adapting the application method to the different requirements. Experimentation with the recognition system was necessary to gain the knowledge needed for the application specific rules. This is an ongoing process as the system evolves, with the user providing the learning element for the system. The rule base also changes as new applications

```
(def-class requirements nil
                   :slots (name
                           task
                           time
                           space
                           accuracy
                           complete
                           occlusion)
                   :mv (applications))
```

```
Definition for Requirement class instance {712}:

{712} has the following values:
(REQUIREMENTS NAME) --> 712
(REQUIREMENTS TASK) --> RECOGNITION
(REQUIREMENTS TIME) --> 10
(REQUIREMENTS SPACE) --> 9
(REQUIREMENTS ACCURACY) --> 1
(REQUIREMENTS COMPLETE) --> YES
(REQUIREMENTS OCCLUSION) --> NO
(REQUIREMENTS APPLICATIONS) --> (LFF GLOBAL)
```

Figure 13: The Requirements FROB Class and Sample Instance

```
{"LFF-712"} has the following values:
(SYSTEM-SPECS NAME) --> "LFF-712"
(SYSTEM-SPECS TASK) --> RECOGNIZER
(SYSTEM-SPECS REQUIREMENTS) --> {712}
(SYSTEM-SPECS METHOD) --> LFF
(SYSTEM-SPECS TIME) --> 10
(SYSTEM-SPECS SPACE) --> 9
(SYSTEM-SPECS ACCURACY) --> 1
(SYSTEM-SPECS CAMERA) --> {CCD-CAMERA}
(SYSTEM-SPECS CAD-IMAGES) --> {HINGE}
(SYSTEM-SPECS MODEL) --> {"MODEL-LFF-712"}
(SYSTEM-SPECS MODELER) --> BUILD-LFF-MODEL
(SYSTEM-SPECS RECOGNIZER) --> *UNDEFINED*
(SYSTEM-SPECS RLE-CONVERTER) --> {RLE-TO-GRAY}
(SYSTEM-SPECS IKS-FORMATTER) --> *UNDEFINED*
(SYSTEM-SPECS FEATURES) --> ({HOLE-FINDER} {CORNER-FINDER})
(SYSTEM-SPECS ROBUST-FEATURES) --> *UNDEFINED*
(SYSTEM-SPECS FF-PRIORITY) --> ((TYPE HOLE) (TYPE CORNER))
(SYSTEM-SPECS CLUSTER-SIZE) --> 4
(SYSTEM-SPECS REC-SIZE) --> 3
(SYSTEM-SPECS REL-ORIENT-VAR) --> 0.3
(SYSTEM-SPECS DIST-VAR) --> 0.1
(SYSTEM-SPECS VALUE-VAR) --> 0.3
(SYSTEM-SPECS SEGMENTER) --> {SEGMENTER}
(SYSTEM-SPECS MOMENTER) --> *UNDEFINED*
(SYSTEM-SPECS DISCRIMINATOR) --> {MAX-CLIQUE-FINDER}
(SYSTEM-SPECS SHOWER) --> {DISPLAY-ON-ELAN}
(SYSTEM-SPECS DISPLAYER) --> {OBJECT-OUTLINER}
(SYSTEM-SPECS COMPLETE) --> SET
```

Figure 14: System Specification Instance Created by Application Selection Rules

are provided. The vision system has been designed so the new rules should not interfere with existing rules. They can have an effect, however, in the application selection knowledge base if there is a restriction on how many applications can be considered in parallel.

5.2 The Global Recognition System

Initially the prototype vision system was created with a global recognition application system. The segmentation, feature extraction, and model matching code for the application existed on a Vax written in C. The intention for the early system was to implement this recognition application using the object-based approach of the vision system. To do this, class FROBs were defined to represent the algorithms, hardware and sensors. Methods and Lisp functions were written to execute and operate the software and sensors and pass information between them. The first global application system had little flexibility in its operational parameters, so the knowledge needed to configure the application was highly structured and done with Lisp functions rather than rules. Figure 15 shows the configuration of the global application system. This configuration remained constant since it reflects the general concept of model-based visual recognition.

Object models to test the system were designed using Alpha-1, and binary image renderings of these models were used for training. The system interface to Alpha-1 consisted of an automatic renderer which produced three renderings at preset rotations of the modeled object. The viewing angles selected provided the greatest variance in feature values; 0, 12 and 45 degrees. Each rendering was converted to a gray level image which could be segmented and processed by the application algorithms. Training consisted of extracting all available features from the images and obtaining a variance for each. Figure 16 shows a sample object rotated in the three positions.

Table 2 shows the corresponding features for each rotated rendering of the model object.

Each feature's variance was used to determine its robustness which determined if it was selected as a model feature. The number of features selected for the model corresponded to the speed and accuracy requirements. Weights were calculated for each model feature using these variances.

The trainer also produced a logical sensor object instance which controlled all of the algorithm and sensor instances created for the system. The logical sensor represented the recognition system at its highest level and when combined with the model the application system is complete. The system is operated by using a method to send a message to the logical sensor requesting output from it. This in turn causes the logical sensor to run its program which operates the application system.

Several model objects were created using Alpha-1 and the NC milling machine to test the global recognition application system. The sample part shown in the figures was milled from wax so it could be placed in front of a camera for analysis. It took the application system trainer less time to create a model and recognizer for the sample object than it did to mill it, which was on the order of 10-15 minutes. The recognition system contained the same algorithms which were used during training for consistency. Since object instances were created to represent the particular recognition system, more than one model and recognizer could exist within the system. These recognition systems were tested by placing several objects on a backlit table from which an image

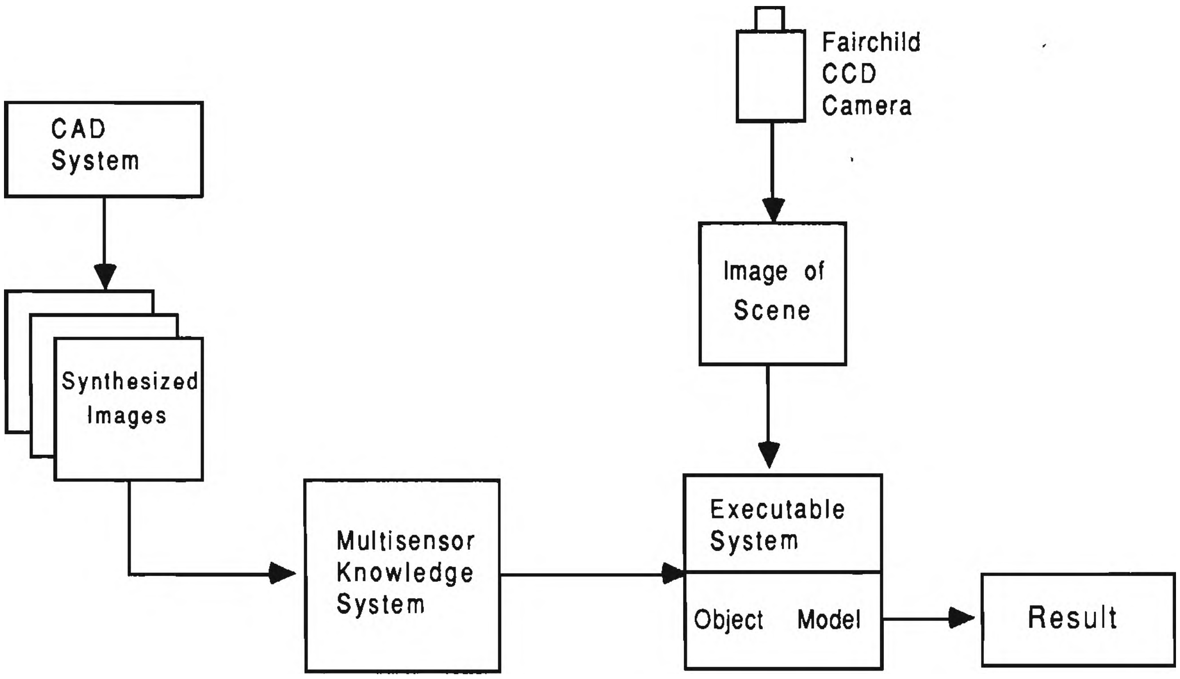


Figure 15: Global Feature Recognition System

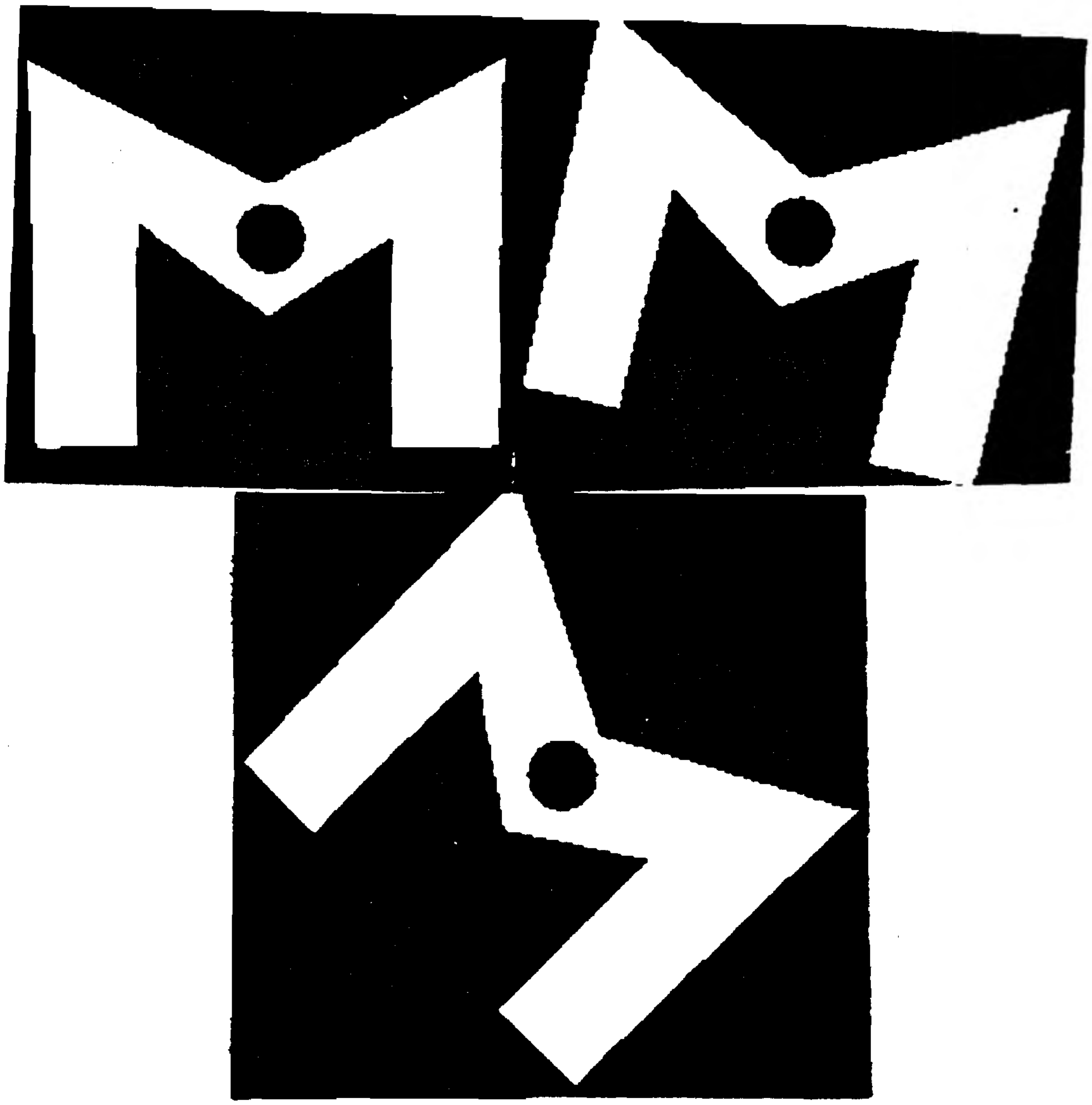


Figure 16: Rotated Renderings of a Sample Object

Table 2: Global Training with Model Features and Their Corresponding Weights

Feature	0 degrees	12 Degrees	45 Degrees
area	15037.0000	15035.0000	15042.0000
perimeter	1084.0000	1166.0000	1258.0000
diameter	180.0000	203.0000	230.0000
thinness	78.1443	90.4261	105.2097
p_a	0.005197	0.006014	0.006994
x_y_inertia	0.832402	0.891089	1.000000
holes	1	1	1
n1_moment	20.667648	17.377649	20.211788
n2_moment	20.837383	20.555441	20.479317
n3_moment	21.207891	21.342537	21.375788
n4_moment	21.206545	20.074003	20.006145
n5_moment	20.985352	21.419800	20.308971
n6_moment	19.279917	20.586561	20.204609
n7_moment	20.562645	21.247772	21.297676

Feature	Mean Value	Weight
area	15038	0.958694
n2_moment	20.624048	0.027684
n3_moment	21.308741	0.013622

could be extracted and analyzed. The application system was then invoked and the result analyzed for validity.

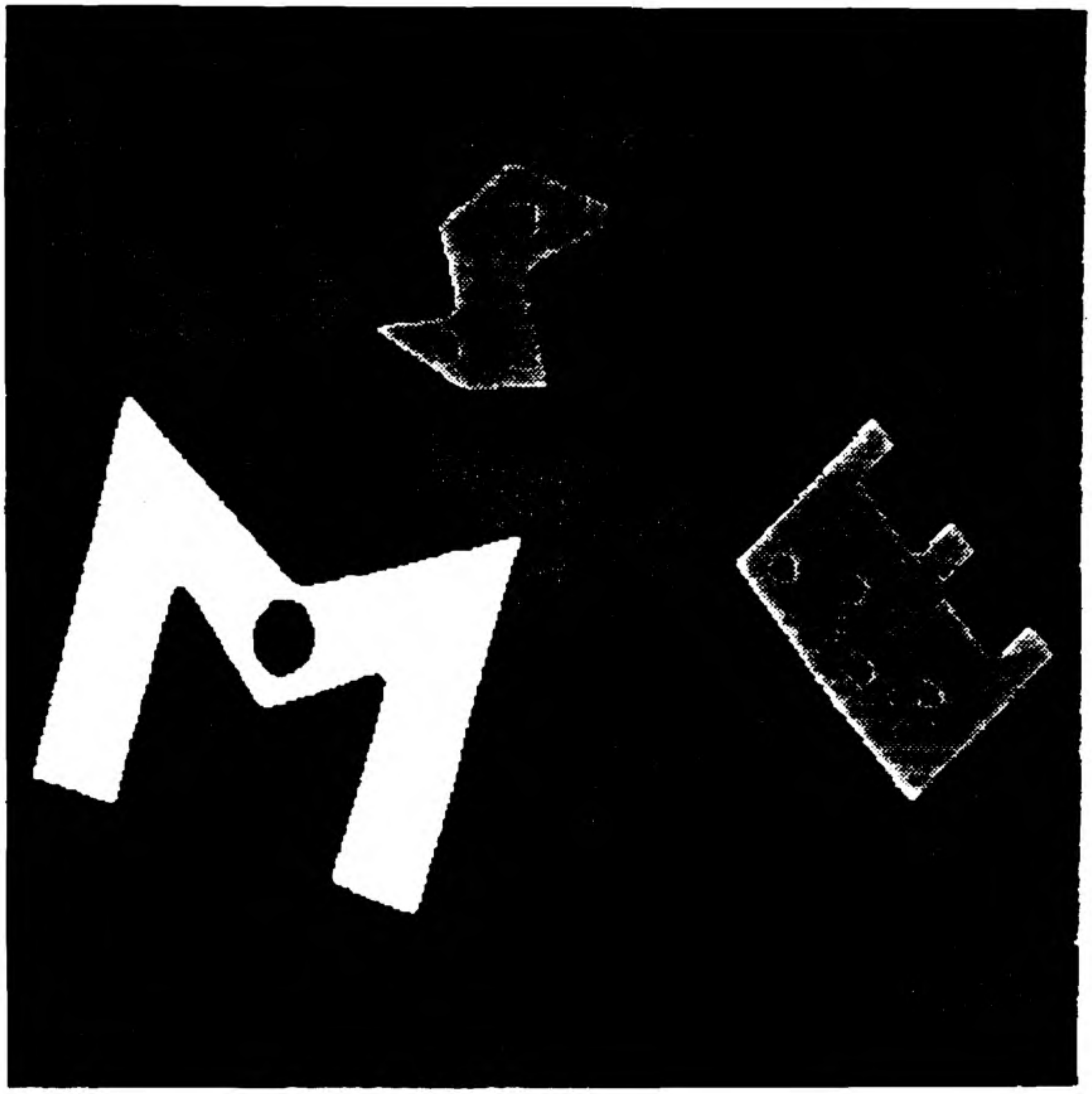
Figure 17 shows a scene used to test the recognition system and model of the sample object. It also shows the features and distances calculated for each object in the scene, the smallest distance being the wax representation of the sample object meaning the system was successful when used on this particular scene.

The global application system was highly successful for the objects tested. The trainer was set with a threshold which selected between 5 and 7 robust features per model. Processing time was reasonable considering that images of size 480 rows by 512 columns were processed: the size of the images transmitted from the camera and digitized by the Vicom. The synthesized recognition systems usually took from 3 to 5 minutes to process scenes, most of which contained three objects of a similar size to the example shown. The speed of the system largely depended on the system load and ethernet traffic at the time of operation, and the number and size of objects on the scene. Weaknesses in the application method itself prevailed as the Area feature provided an overwhelming amount of robustness in comparison with other global features. Usually the object's area was weighted at 90 percent of the distance calculation showing the lack of robustness in the global method as a whole. This weakness caused one erroneous matching of an object to a model as a result.

The global application system's flexibility in configuration evolved with the increased functionality of the knowledge-based system as a whole. The first global system tested had all of the vision algorithms written in C on the Vaxen with only the result being transferred to the Lisp environment on the Bobcat. Later applications had the benefit of methods created for the Local Feature Focus application system which provided for transparent transfer of information between machines. Further parameterization of the global application system was performed to enable the use of rule knowledge. The recognizer was parameterized so all thresholds and tolerances could be set as values in the slots of algorithm objects. These parameters included: features to be considered during training, the variance tolerance to select robust features, and the distance threshold for recognizing objects. In the final global application system created, the results, which are seen in several of the previous figures, use the full power of the knowledge-based system with requirements and rules replacing the Lisp functions used to synthesize the recognition systems. The rules added some flexibility in how the recognizers were configured but the over robustness of the "area" feature restricted the variety of recognition systems that could be created by the rules. A system using area alone could usually be used to meet any possible set of requirements. The application method did prove invaluable, however, in development of the knowledge-based system concept as a whole by providing a specific example for experimentation purposes.

5.3 Local Feature Focus

Local Feature Focus was added as a recognition method available to the knowledge-based vision system. It was chosen for its simplicity and robustness as a relational graph method and provided the vision system with additional capabilities such as occluded object recognition. Implementation



```
3 Objects found in scene.
Distances for objects:

((1 0.774031) (2 0.139606) (3 0.519891) )
```

Figure 17: Object Recognition on a Sample Scene

of the application required the writing of local feature extraction algorithms, more specifically hole and corner detectors. They were both written in C which best suited their speed requirements. Other algorithms to perform the model building and graph matching were written in Lisp which is better suited for their requirements. Unlike the global method which had minimal interaction between the C and Lisp environments, Local Feature Focus requires frequent passing of information between algorithms in both environments. This provided the opportunity to test system methods which perform this utility for the vision system. The addition of this application to the vision system also made it possible to test several application selection rules since more than one method for recognition has become available to the system.

5.3.1 Local Feature Detection

Detection and localization of local features is a more difficult task than extracting global features. This is due to the more sophisticated nature of the local features verses the statistical nature of the global features. The extraction of local features from an image can be viewed as a search rather than a calculation involving a group of pixels. The image is searched for a group of pixels having an organization of interest. More specifically, for Local Feature Focus, the image was searched for instances of holes and corners.

Hole detection can be done using the same algorithm used for the segmentation of objects within an image. Holes within an image can be looked upon as objects within an object, and can be segmented out of the image as a whole. The segmentation algorithm was modified to segment all connected regions of background surrounded by an object. Global properties of the holes detected are used to attach values to the feature, such as its area or center of mass as its location. A feature class instance is created for each hole detected; its slots containing specific information about the feature. Figure 18 shows a feature object instance for a hole. All features extracted from an image are stored as objects within the system for the training and recognition algorithms to use. Once all local features have been extracted they are classified so they can be used to build the focus feature cluster model. Feature classes are always defined for each feature type exclusively, holes and corners can never belong to the same class. Features of a common class are defined by a value within a given range. The range is determined by a threshold for the feature's value variance from a previously detected feature value. These threshold values are predefined for the system according to the accuracy of the feature extraction algorithms. A low threshold value will produce more feature classes thus making the model less complex and less accurate for complex objects. A high threshold value may cause a problem in grouping too many features together thus making the recognition method less effective overall.

Corner detection within an image is unfortunately not as easy to implement as hole detection. Corners are defined as a sharp transition in an objects perimeter. If we are dealing strictly with polygons, corners will always be located at the connection point of its line segments. It therefore is logical to extract corners for an object's model from the CAD representation of the object itself since this information should be readily available. This capability was not used for this vision system, rather a corner detector was written for use with gray level images. This gray level corner detector had to be implemented regardless of how the trainer obtained the corners for the object

```
{"HOLE5"} has the following values:
(FEATURE NAME) --> "HOLE5"
(FEATURE TYPE) --> HOLE
(FEATURE LOCATION) --> (296 272)
(FEATURE ORIENTATION) --> *UNDEFINED*
(FEATURE VALUE) --> 55
(FEATURE CLASS) --> {CLASS-FEATURE 1}


{CLASS-FEATURE 1} has the following values:
(FEATURE CLASS) --> *UNDEFINED*
(FEATURE VALUE) --> 52
(FEATURE ORIENTATION) --> *UNDEFINED*
(FEATURE LOCATION) --> *UNDEFINED*
(FEATURE TYPE) --> HOLE
(FEATURE NAME) --> *UNDEFINED*
(CLASS-FEATURE COLOR) --> *UNDEFINED*
(CLASS-FEATURE VALUE-VARIANCE-PERCENT) --> 0.3
(CLASS-FEATURE VALUE-VARIANCE) --> 15.6
(CLASS-FEATURE CHORD-LENGTH) --> *UNDEFINED*
(CLASS-FEATURE SECONDARY-FEATURES) --> *UNDEFINED*
(CLASS-FEATURE SECONDARY-VARIANCES) --> *UNDEFINED*
```

Figure 18: A Feature Class Instance for a Hole and Its Class Definition

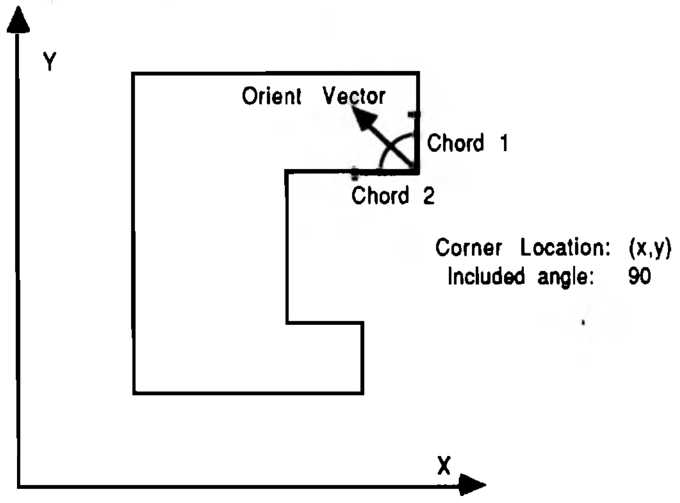


Figure 19: Corner Detection in an Object

model. This is true since corners must be extracted by the object recognizer from camera images too.

A significant amount of work has been done seeking an efficient way to detect corners of an object within an image. Bolles described a method used in the SRI vision module implementation of local feature focus [5] which is a fast and fairly reliable method of extracting corners of an object from an image. It uses a pair of jointed chords to trace around the perimeter of the object a pixel at a time. The length of the two chords is maintained at a constant Manhattan distance. As the joint or center of the two chords moves about the object, the chords form an inclusive angle less than 180 degrees. The algorithm measures the rate of change in the angle hoping to detect a corner when the rate of change swings from negative to positive. The angle of the corner once detected is calculated by using a bisector of the chord angle to determine object orientation. If the bisector points inward to the object, the angle of the chords is used; otherwise it is subtracted from 180 degrees. Corner orientation is taken as the angle of the bisector of this corner angle.

The accuracy of this method depends primarily on two factors. If the length of the chords is too short, noise and false corners will more readily be detected. If the chords are too long, false angles may be reported for some corners. It was mentioned in the technical report that a straight line distance function provided more robustness in angle detection but increased the calculations required significantly. This method was used for the knowledge base application for testing but both are provided to give some flexibility in meeting the time and accuracy requirements of the system. Figure 19 shows graphically how a corner is detected in an object.

The corner detector was fine tuned to provide robustness in its angle detection. This involved defining certain tolerances for the method and finding what values assigned to them work best. If every change in chord angle from negative to positive were reported as a corner, many false corners result from a camera image due to noise and quantization errors. A tolerance was set for how much the angle between the chords must change before a corner is to be detected. Another tolerance was set for the chord angle itself; this eliminated detection of smooth curves, only angles less than the tolerance are reported as corners. These tolerances were adjusted to remove most of the false corners detected in the camera images. The rate of false detection with the tolerances used was on the order of 1 in 15 which is about 7 percent, an acceptable rate. Figure 20 shows an image of a CAD designed object from which holes and corners have been extracted and labeled for use by the model builder.

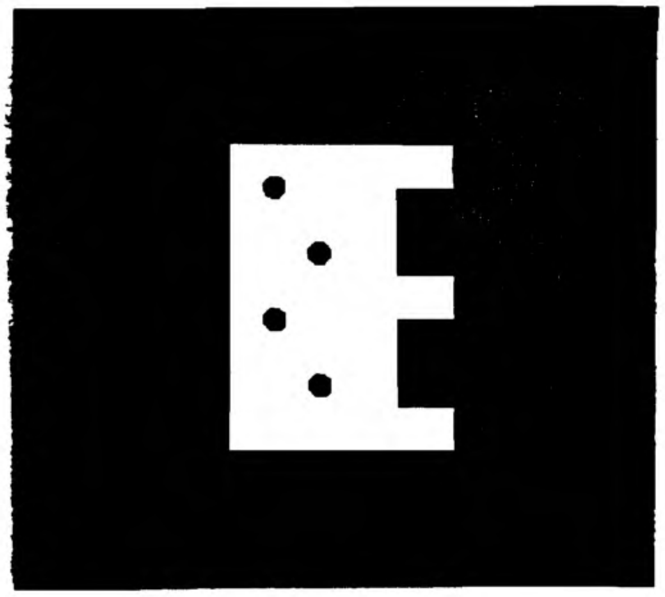
5.3.2 Building a Local Feature Focus Model

The model building part of the local feature focus application was written in Lisp. Lisp functions perform all the training for the application. This involves the manipulation of values produced by the feature extraction algorithms to create a model class instance which represents the model. The trainer uses this information in conjunction with information stored in the system specification object.

The Local Feature Focus trainer builds a list of focus feature clusters, each of a size determined by the system specification. Calculations are performed to determine relative distances and orientations between focus and secondary features in each cluster. This information is organized and stored as a list in a slot of the model instance.

A boundary list for the object is also created by the trainer using the corners extracted from the CAD image. The hypothesis verification method, which is discussed later, is used to determine the transitions along the boundaries of the model object. These transitions are recorded as "T,NIL" pairs, "T" representing the object and "NIL" representing the background. The pair represents the transition from either object or background to the same. The pair "(T NIL)" would represent a transition from object to background at the current point along the boundary in the image. Measurements are taken at ten evenly spaced points along any line within the boundary by an algorithm which passes the transition information to the trainer to be placed in the model. Figures 21 and 22 show a model instance created by the local feature focus trainer using the CAD object and system specifications shown earlier. The priority of the focus features selected was holes followed by corners. Only clusters for the holes and 270 degree corners (class 3, 4.712 radians) are shown due to space considerations. The names given to the feature instances are generic and generated by the system at run-time.

Only slot information pertinent to the Local Feature Focus recognition method is defined in the example; undefined slots are used by other recognition methods or have not been implemented. The two numbers following the secondary features in the cluster information are the relative distance and orientation of the secondary feature to the focus feature, respectively. The model shown in the figure does not include the boundary transition pairs.



Name	Location (X Y)	Included Angle (Radians)	Orientation (Radians)
CORNER117	(214 272)	1.570796	0.785398
CORNER118	(214 156)	1.570796	-0.785398
CORNER119	(296 156)	1.570796	-2.356194
CORNER120	(296 172)	1.570796	2.356194
CORNER121	(275 172)	4.712389	2.356194
CORNER122	(275 206)	4.712389	-2.356194
CORNER123	(296 206)	1.570796	-2.356194
CORNER124	(296 222)	1.570796	2.356194
CORNER125	(275 222)	4.712389	2.356194
CORNER126	(275 256)	4.712389	-2.356194
CORNER127	(296 256)	1.570796	-2.356194
CORNER128	(296 272)	1.570796	2.356194
Name	Location (X Y)	Hole Area (Pixels)	Orientation (Radians)
HOLE111	(230 256)	52	None
HOLE112	(247 231)	52	None
HOLE113	(230 205)	55	None
HOLE114	(247 180)	54	None

Figure 20: An Image of a CAD Object With its Local Features Extracted

```
{"MODEL-LFF-712"} has the following values:
(MODEL NAME) --> "MODEL-LFF-712"
(MODEL TYPE) --> LFF
(MODEL FEATURES) --> ({"HOLE111"} {"HOLE112"} {"HOLE113"}
                      {"HOLE114"} {"CORNER117"} {"CORNER118"}
                      {"CORNER119"} {"CORNER120"} {"CORNER121"}
                      {"CORNER122"} {"CORNER123"} {"CORNER124"}
                      {"CORNER125"} {"CORNER126"} {"CORNER127"}
                      {"CORNER128"})
(MODEL SAMPLES) --> *UNDEFINED*
(MODEL TRAINER) --> LFF-TRAINER
(MODEL BOUNDARY-LIST) --> ({"CORNER117"} {"CORNER118"}
                           {"CORNER119"} {"CORNER120"}
                           {"CORNER121"} {"CORNER122"}
                           {"CORNER123"} {"CORNER124"}
                           {"CORNER125"} {"CORNER126"}
                           {"CORNER127"} {"CORNER128"})
(MODEL FEATURE-WEIGHTS) --> *UNDEFINED*
(MODEL CLASS-FEATURES) --> ({CLASS-FEATURE 1}
                            {CLASS-FEATURE 3}
                            {CLASS-FEATURE 2})
(MODEL DISTANCE-CALCULATOR) --> *UNDEFINED*
(MODEL FOCUS-FEATURES) --> *UNDEFINED*
(MODEL CLUSTER-SIZE) --> 3
(MODEL VALUE-VAR) --> *UNDEFINED*
(MODEL REL-ORIENT-VAR) --> 0.3
(MODEL REC-SIZE) --> 4
(MODEL DIST-VAR) --> 0.1
```

Figure 21: LFF Model Instance for CAD Object (Part I)

```
(MODEL LOCAL-CLUSTERS) -->
(({CLASS-FEATURE 1}
  ({"HOLE111"}
   (({"CORNER117"} 22.627416997969522 0)
    ({"HOLE112"} 30.23243291566195 0)
     ({"CORNER126"} 45.0 0)))
  ({"HOLE112"}
   (({"CORNER125"} 29.410882339705484 0)
    ({"HOLE111"} 30.23243291566195 0)
    ({"HOLE113"} 31.064449134018133 0)))
  ({"HOLE113"}
   (({"HOLE114"} 30.23243291566195 0)
    ({"HOLE112"} 31.064449134018133 0)
    ({"CORNER122"} 45.0111097397076 0)))
  ({"HOLE114"}
   (({"CORNER121"} 29.120439557122072 0)
    ({"HOLE113"} 30.23243291566195 0)
    ({"CORNER122"} 38.2099463490856 0))))
 ({CLASS-FEATURE 3}
  (({"CORNER121"}
    (({"CORNER120"} 21.0 0.0)
     ({"CORNER119"} 26.40075756488817 4.712388)
     ({"HOLE114"} 29.120439557122072 0)))
   ({"CORNER122"}
    (({"CORNER125"} 16.0 4.712388)
     ({"CORNER123"} 21.0 0.0)
     ({"CORNER124"} 26.40075756488817 4.712388)))
   ({"CORNER125"}
    (({"CORNER122"} 16.0 4.712388)
     ({"CORNER124"} 21.0 0.0)
     ({"CORNER123"} 26.40075756488817 4.712388)))
   ({"CORNER126"}
    (({"CORNER127"} 21.0 0.0)
     ({"CORNER128"} 26.40075756488817 4.712388)
     ({"CORNER125"} 34.0 4.712388))))))
(MODEL MACHINE) --> *UNDEFINED*
(MODEL FILE-NAME) --> *UNDEFINED*
```

Figure 22: LFF Model Instance for CAD Object (Part II)

The recognition function uses the same local feature extraction algorithms along with the model to build a graph of consistent model/image node pairings. A system specification parameter determines the minimum size of a clique in the graph to be considered as a hypothesis. This parameter has the effect of determining how selective the recognizer is towards consistency between nodes. A high threshold will cause more false hypotheses to be considered on a percentage-wise basis. This in general causes greater overhead in determining which hypotheses are correct. A threshold set too low may cause problems with correct hypotheses being ignored due to the effects of noise or a slightly imperfect part, but should cause the system to perform more rapidly since fewer hypotheses will be verified. It becomes obvious that this parameter can be used to tune the system to meet speed and accuracy performance requirements.

The hypothesis verifier was written in C using the same method described by Bolles. It uses a rotated and translated polygonal outline of the object to check for consistent transitions with the model. The translation of the outline was extracted by subtracting the model feature locations from the image feature locations and averaging them. The rotation of the outline is then calculated by taking the two most distant image features in the clique and using the change in angle of a line drawn between them for both the model and image. Originally corner feature orientation offsets were used to obtain this angle but were found to be highly inaccurate in general and they also eliminate the possibility of using a clique consisting entirely of holes since they do not have orientations. This method for verifying hypotheses was derived from the method described in a paper on Local Feature Focus by Bolles and Cain [4].

The method was tested on several scenes with a high degree of success. The output of the recognizer is the list of translated and rotated points of the template which has been used for verification of the hypothesis. To graphically verify that the hypothesis is reasonable, a display function was added to highlight the outline created by these points within the scene image and display it on the Bobcat frame buffer. Figure 23 shows displayer output for the Local Feature Focus method used with the object model shown in the previous figure.

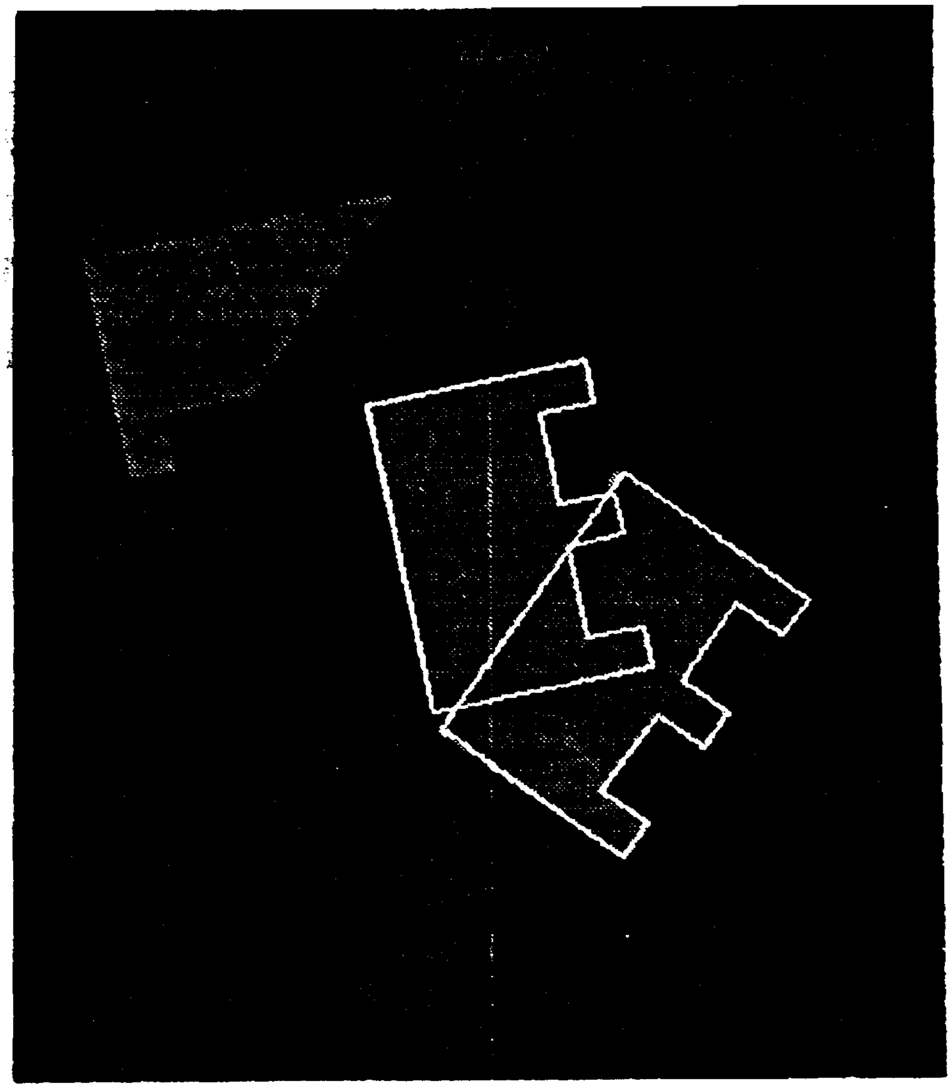


Figure 23: Output for the Local Feature Focus Analysis of a Scene

5.4 Local Feature Inspection

An inspection application was added to the knowledge-based system which provides a model-based inspection capability. The feature detection algorithms used for the Local Feature Focus application are used to build the inspection model. This model consists of a list of features, their positions, values and orientations. The application task is to report on the accuracy of an object, the amount of error in the position, value and orientation of each local feature. This is done by extracting the local features in a known order from an image of the object to be inspected. This list of features is compared to the list of features in the model to produce the inspection output.

The inspection application requires that only one instance of the object to be inspected appears in the image. The object must also be in a similar orientation as it was when the inspection model was created, so features are extracted in the same order as done for the model. Missing features are reported as well as a list of differences in position, value, and orientation of those found. This information is displayed in a table format to the user but can also be used as input to another application which might accept or reject the object according to preset tolerances.

Several objects were used to test the inspection system. Figure 24 shows an image of a defective version of the hinge shown earlier in the Local Feature Focus example. The output of the inspection system reports the missing holes and the differences in the corner angles and orientations as compared with the hinge model as seen in Figure 25.

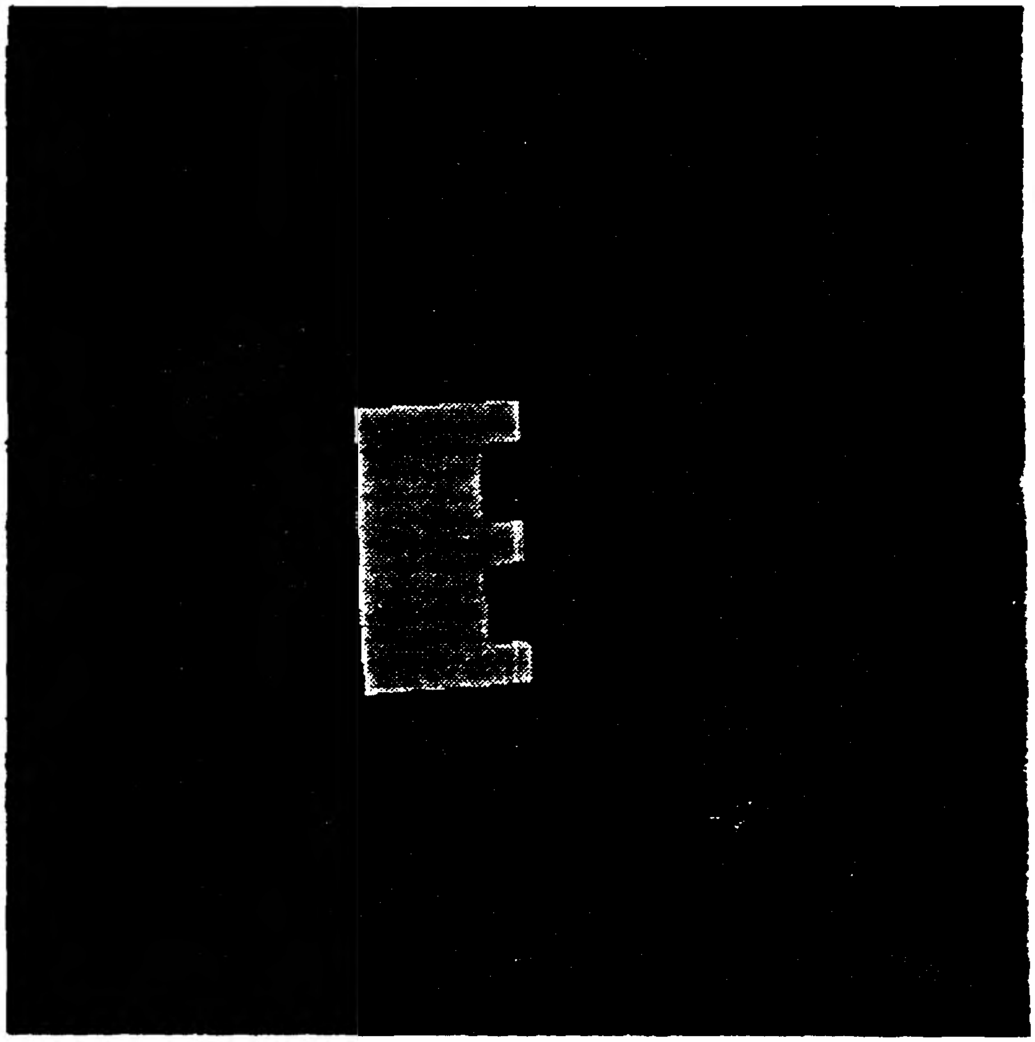


Figure 24: Image of a Defective Hinge to be Inspected

```
Feature HOLE168  dloc: NOT-FOUND     dval: NIL     dorient: NIL
Feature HOLE169  dloc: NOT-FOUND     dval: NIL     dorient: NIL
Feature HOLE170  dloc: NOT-FOUND     dval: NIL     dorient: NIL
Feature HOLE171  dloc: NOT-FOUND     dval: NIL     dorient: NIL
Feature CORNER174  dloc: (39 -2)     dval: 0.896   dorient: -1.030
Feature CORNER175  dloc: (-16 111)   dval: 0.0     dorient: -1.570
Feature CORNER176  dloc: (-94 3)     dval: -0.197  dorient: -1.405
Feature CORNER177  dloc: (-33 -8)    dval: 0.0     dorient: -4.712
Feature CORNER178  dloc: (-14 7)     dval: -2.987  dorient: -0.266
Feature CORNER179  dloc: (-25 -28)   dval: -0.463  dorient: -4.248
Feature CORNER180  dloc: (-51 -1)    dval: 2.677   dorient: -0.321
Feature CORNER181  dloc: (-36 -12)   dval: 0.197   dorient: -4.712
Feature CORNER182  dloc: (-15 3)     dval: -3.338  dorient: -0.165
Feature CORNER183  dloc: (-30 -32)   dval: -0.165  dorient: -4.621
Feature CORNER184  dloc: (-51 -2)    dval: 2.779   dorient: -0.110
Feature CORNER185  dloc: (-37 -15)   dval: 0.156   dorient: -4.978
NIL
```

Figure 25: Inspection Output for Defective Hinge

6 Discussion

6.1 Application Performance

The knowledge based system was tested with three application systems with mixed results. The global feature recognition system was the first to be implemented. As previously stated the initial global recognition system used no rules and had its operational parameters preset. An attempt was made to select parameters to vary the system's performance such as the variance tolerance for selecting robust features for the model. This was done with little success since the area feature was always weighted at over 90 percent. The system did however perform well on the objects tested and provided the framework for developing other applications.

The Local Feature Focus system produced the best results, yet it also proved to be highly constrained in its performance. This method also had a small rule set since an optimal configuration was obtained which worked well for most application situations. Further testing of the system may provide additional knowledge about adjusting the application's performance. The application performed quite well in several difficult situations involving occlusion and parts with similar features to the model.

6.2 Hardware Configuration Performance

The knowledge-based system is implemented with a hardware configuration that has several constraints. The most prominent constraint is that the knowledge base exists on the HP Bobcat which is the hub of the hardware configuration. This configuration is the only one possible which allows the knowledge-based system to control remote algorithms and sensors without creating special purpose system software. All remote tasks are performed from the HP Bobcat using existing high level Unix remote machine utilities. The rcp utility provides the capability to copy files between two hosts from another host, which prevented the HP Bobcat from becoming a bottleneck for interprocessor dataflow. The rsh utility allowed the knowledge-based system to execute and direct I/O for programs on remote machines. The use of these utilities to control remote processing restricts the remote processors to be Unix machines connected to the ethernet. This is not a serious problem for the prototype system since all of the available processors support Unix and are connected to the ethernet. It does however restrict the portability of the knowledge-based system and the hardware it can use. New methods could be written to allow the execution of programs on non-Unix machines but currently no such capability exists.

6.3 Knowledge System Summary

The knowledge-based vision system provides a powerful and flexible tool for the development of vision applications. The hardware and software design of this prototype system has some restrictions but these are outweighed by the benefits. The inspection application was designed using components from the Local Feature Focus recognition system proving the flexibility of the system to create new applications from those existing. The modular construction and operation of applications provided excellent debugging capabilities since each application component could be isolated for testing. The underlying FROBS system was sufficiently powerful to meet the needs of the knowledge-based system. Overall the prototype knowledge-based system was a success in providing a useful tool for vision application system design. There is room however for improvement in several areas of its operation. The next sections discuss some possibilities for future work on the system.

6.4 Application Expansion

The knowledge-based system can support a large number of applications. Two recognition applications and one inspection application were tested in the prototype system. Modification of these applications may provide more flexibility in their operation and add power to the knowledge-based system as a whole. Additional inspection applications can be created using global features rather the local features used for inspection in the prototype system. New vision applications can be synthesized from the existing algorithms in the knowledge base. There is room to create many new applications from existing vision application system components.

6.5 Expansion of Requirements Specification

With new applications made available to the knowledge-based system the requirements are expanded as well. New slot values are required to cause new rules to fire to configure the application. It would be highly desirable to provide a friendlier requirements specification interface for the user. A menu driven system would be adequate, providing information regarding what range of values is permitted for different requirement parameters.

6.6 CAD Interface Expansion

Presently the knowledge-based system has a fixed interface to the CAD system. It uses shell scripts which issue commands to provide renderings and their respective run length encoded files as input for training. Eventually this would prove to be inadequate if an application demands more information directly from the CAD model rather than a rendered image. The local feature focus application could conceivably extract its local features from the CAD model representation directly. This would only be advantageous if the local features are primitives used by the modeling system or the features can be more accurately computed given exact mathematical representations of perimeter edges or holes in the object. Values from these exact representations are perfect in that no quantization error exists, but do not represent what the camera will eventually see.

6.7 Application Verification Rules

As mentioned earlier, it would be desirable to have a set of rules govern the verification process of the applications created by the system. More specifically a rule base could be developed to verify that the applications created perform within the parameters of their requirements. This set of knowledge would use simulation and software verification tools to perform its task. The rules would activate functions to perform the benchmarking of the application and then act upon the results to either reject the application as insufficient in meeting the requirements or accept the application as suitable. They would also generate new rules or eliminate old rules from the existing rule base if simulation results prove current knowledge inaccurate.

These rules are not essential to the vision system as a whole but add another level of automation to the system. Eventually a vision system containing many applications would need some sort of automated verification capability.

6.8 Automated Learning

Some advanced expert systems incorporate an automated learning function into the knowledge base [10]. New knowledge can be synthesized from the analysis of applications produced by the knowledge-based system. This could conceivably be done by the verification rules which simulate the operation of the application and take several benchmark measurements to determine application efficiency. New rules can be synthesized from this information and checked for consistency with existing knowledge. Conflicts may cause the system to update its current knowledge and nonconflict knowledge would simply be added to existing knowledge. A function such as this would add intelligence to the system since it would react intelligently to its environment. Development and testing of the learning function would be difficult but possible. Figure 26 shows a diagram of the general components and their functions for a learning function.

6.9 Parallel Implementation

The current object system, FROBS, runs in a nonconcurrent implementation of Common Lisp. For the full parallel potential of the knowledge-based vision system to be realized, a concurrent Lisp would have to be used. This would provide allocation of independent tasks in the system to separate processors. Application selection and specific rules could run in parallel and application systems could be synthesized in parallel as well. For the application itself to operate in parallel is a difficult problem. Parallel implementation of the application becomes another knowledge-based task where many more factors exist than in the sequential application. Knowledge of how to implement the application in parallel would need to be supplemented with new methods that provide object to object communication and coordination in a multiprocessor environment.

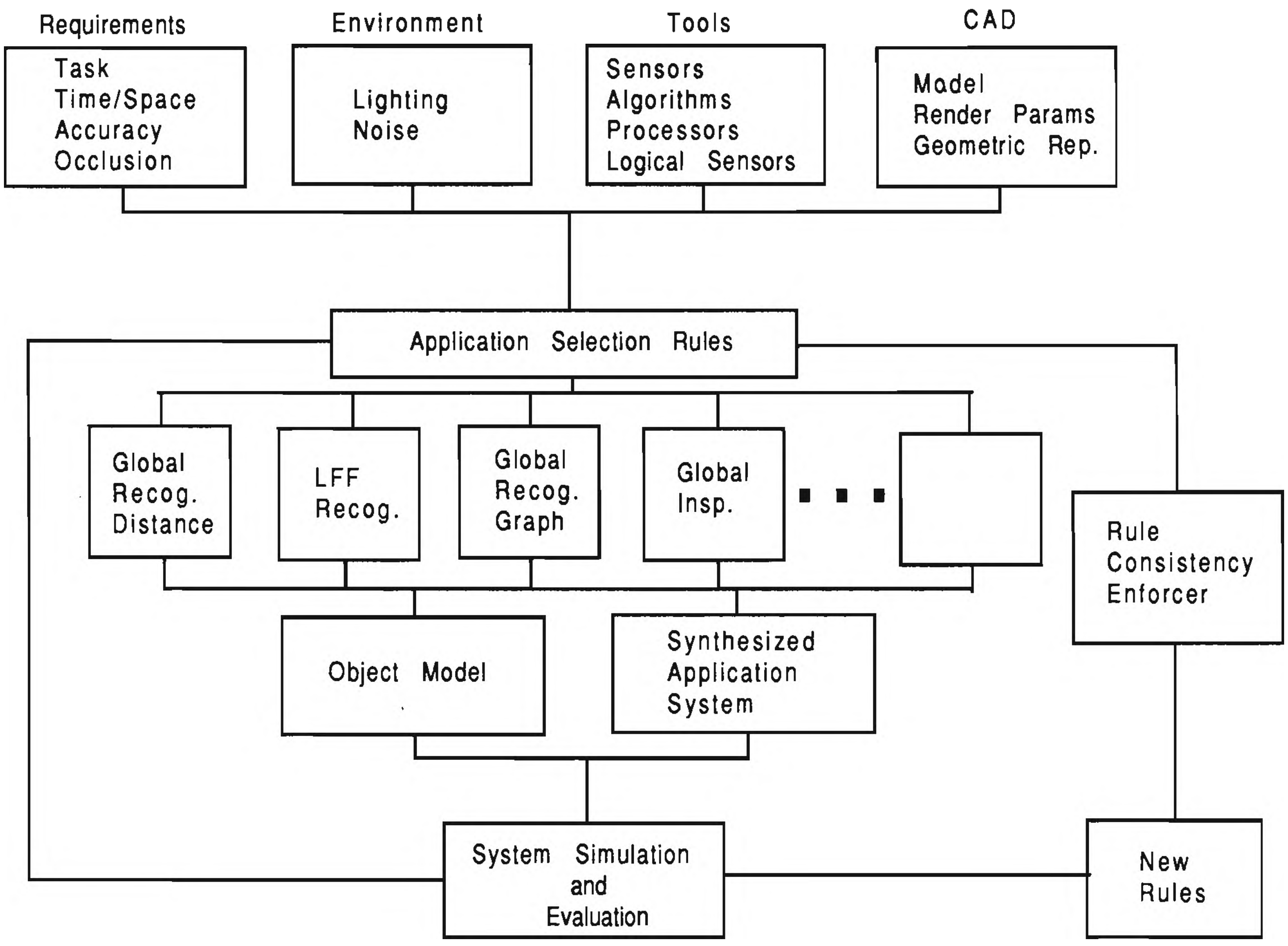


Figure 26: Adding a Learning Component to the Knowledge-Based System

A FROB Class Definitions and Methods for Algorithms

```
(def-class algorithm nil
           :slots (name
                   size
                   language
                   machine
                   executable
                   args
                   input
                   input-type
                   output
                   output-type
                   e-function
                   machine-out
                   complexity
                   stability
                   robustness
                   p-factor))

;
; run for algorithm class produces (machine file) as output
;
(def-method ({class algorithm} run) ()
            (let* ((mach (machine $self))
                   (exec (executable $self))
                   (args (args $self))
                   (function (e-function $self))
                   (genout (conc (conc "/tmp/"
                                       (string (name $self)))
                                 (string (gensym))))
                   (out-type (output-type $self))
                   (in-type (input-type $self))
                   (genin "/tmp/stdin")
                   (input (input $self))
                   output)
      ; if output exists, do nothing
              (cond ((def (output $self)) nil)
                    ((def exec)     ; if executable provided
     ; if file as input
                     (cond ((equal in-type 'file)
   ; if more than one file
                                      (if (listp input)
                                        (setf input  ; combine them
                                              (cat-input-files
                                               mach input))))
  ; if not file
                                     (T (put-file mach genin input)
     ; place input sexp in file
                                        (setf input genin)))
                               (system (cond ((def mach)  ; if remote machine
   ;run remote shell
                                              (list "remsh" mach exec
                                                    (if (def args) args)
                                                    '" '< '" input '" '> '"
                                                    genout))
                                             (T (list exec ; else run locally
                                                      (if (def args) args)
                                                      '< input '> genout))))
    ; put output in output slot
                               (setf (output $self)
  ; file output
                                     (cond ((equal out-type 'file) genout)
; sexp output
                                           (T (get-file mach genout)))))
   ; if file as input
                              (T (cond ((equal in-type 'file)
     ; combine input files
                                        (if (listp input)
                                          (setf input
                                                (cat-input-files
                                                 mach input)))
                                        (setf input (get-file mach input)))
                                       (T nil))
                                 (setf output  ; evaluate function
                                       (eval (cons function
                                                   (quote-list input))))
                                 (setf (output $self)
    ; write output
                                       (cond ((equal out-type 'file)
   ; to file
                                              (put-file (machine-out $self)
                                                        genout output)
```

```
                                    genout)
                                   (T output)))))))


(close-class)


(def-class segmenter ({class algorithm})
           :slots (start-index))


;
; Method to make a segmenter class instance
;
(def-method ({class segmenter} make)
            (&rest keys &key start-index &allow-other-keys)
  (let (inst)
    (setf inst (apply-method (algorithm make)
                             :allow-other-keys t keys))
    (when start-index (setf (start-index inst) start-index))
    inst))

(close-class)

(def-class discriminator ({class algorithm}))

(close-class)

(def-class displayer ({class algorithm}))

(close-class)


(def-class feature-calculator ({class algorithm})
           :slots (feature-type
                   focus-type))


;
; Method to make a feature-calculator class instance.
;
(def-method ({class feature-calculator} make)
            (&rest keys &key feature-type focus-type
                   &allow-other-keys)
  (let (inst)
    (setf inst (apply-method (algorithm make)
                             :allow-other-keys t keys))
    (when feature-type (setf (feature-type inst) feature-type))
    (when focus-type (setf (focus-type inst) focus-type))
    inst))

;
; Method to run a feature calculation algorithm
;
(def-method ({class feature-calculator} run-feature) (system-specs)
  (let* ((type (feature-type $self))
         input-frob)
    (cond ((equal type 'standard) ; standard features (non-moment)
           (cond ((def ; check for segmenter output
                   (output (setf input-frob
                                 (segmenter system-specs))))
  ; run feature algorithm
                  (pipe-and-run input-frob $self))
                 (T (format nil
                     "No segmented image provided for input"))))
          ((equal type 'moment) ; if moment feature (normalized)
   ; check for momenter output
           (cond ((def (output (setf input-frob
                                     (momenter system-specs))))
  ; run normalized feature
                  (pipe-and-run input-frob $self))
                 (T (format nil "No moments provided for input"))))
          (T (format nil "Unknown feature type")))))


(close-class)

(def-class distance-calculator ({class algorithm})
           :slots (model-object))


;
; Method to combine feature output for the distance calculator.
;
(def-method ({class distance-calculator} merge-features) (features)
  (setf (input $self) (frob-output features)))
```

```
(close-class)
```

B LFF Instances and Top Level Functions

```
(make {class sensor}
      :name 'ccd-camera
      :type 'visual
      :machine "cs"
      :executable "/u/weitz/vision/run_camera.csh"
      :output-file "/tmp/cam_out.img"
      :output-type 'file)

(make {class segmenter}
      :language 'c
      :name 'segmenter
      :machine "cs"
      :input-type 'file
      :output-type 'file
      :executable "/u/weitz/vision/segment -t 20"
      :start-index 1)

(make {class algorithm}
      :language 'c
      :name 'bi-model-thresholder
      :machine "cs"
      :input-type 'file
      :output-type 'file
      :executable "/u/weitz/vision/threshold")

(make {class algorithm}
      :language 'c
      :name 'object-outliner
      :machine "cs"
      :input-type 'file
      :output-type 'file
      :executable "/u/weitz/vision/outline")

(make {class algorithm}
      :language 'c
      :name 'display-on-elan
      :machine "elan"
      :input-type 'file
      :output-type 'file
      :executable "bin/put_frame -i -b -c")


(make {class discriminator}
      :name 'max-clique-finder
      :language 'lisp
      :e-function 'find-all-cliques
      :output-type 'file
      :machine-out "cs")

(make {class algorithm}
      :name 'rle-to-gray
      :language 'c
      :machine "cs"
      :executable "/s/bin/lss/2D_frob_code/rle_to_gray"
      :output-type 'file)

(make {class algorithm}
      :name 'add-iks-header
      :language 'c
      :machine "cs"
      :executable "/u/weitz/vision/image 480 512"
      :output-type 'file
      :input-type 'file)

(make {class image}
      :name 'hinge
      :size '(480 512)
      :machine "cs"
      :file-name "/u/weitz/vision/hinge_0.img")

(make {class feature-calculator}
      :name 'hole-finder
      :language 'c
      :machine "cs"
      :executable "/u/weitz/vision/find_holes"
      :feature-type 'standard
      :focus-type 'hole
      :robustness 8)

(make {class feature-calculator}
      :name 'corner-finder
      :language 'c
      :machine "cs"
      :executable "/u/weitz/vision/corners"
      :feature-type 'standard
      :focus-type 'corner
      :robustness 10)

(make {class feature-calculator}
      :name 'boundry-finder
      :language 'c
      :machine "cs"
      :executable "/u/weitz/vision/corners"
      :feature-type 'standard
      :robustness 10)

(make {class feature-calculator}
      :name 'boundary-verifier
      :language 'c
      :machine "cs"
      :executable "/u/weitz/vision/verify"
      :feature-type 'standard
      :input-type 'file
      :robustness 10)

;
; Function to train on an image and produce a model and recognizer
; for the local feature focus recognition method.
;
(defun lff-train (system-specs)
  (let* ((name (name system-specs))
         (features (features system-specs))
         (ff-priority (ff-priority system-specs))
         (cluster-size (cluster-size system-specs))
         (value-var (value-var system-specs))
         (rel-orient-var (rel-orient-var system-specs))
         (dist-var (dist-var system-specs))
         (rec-size (rec-size system-specs))
         (verifier (verifier system-specs))
         (clique-tester (clique-tester system-specs))
         (image (cad-images system-specs))
         (modeler (modeler system-specs))
         (model (model system-specs))
```

```
     (rle (rle-converter system-specs))
     (iks (iks-converter system-specs))
     (recognizer (recognizer system-specs))
     (seg (segmenter system-specs))
     x new)
(pipe-and-run image rle) ; convert CAD image to rle
(pipe-and-run rle iks) ; convert rle to iks gray level format
(setf new (make {class model}  ; make a class model instance
                :name (conc "MODEL-" (string name))
                :type 'lff
                :trainer 'lff-trainer))
(clear seg) ; clear segmenter output
(pipe-and-run iks seg) ; run iks image through segmenter
(dolist (x features nil) ; run each feature
  (clear x) ; clear feature output
  (run-feature x system-specs) ; run feature
  ; append feature output
  (setf (features new) (append (features new)
                               (make-feature-frobs x))))
(setf (class-features new) ; sort features by focus priority
      (sort-classes ff-priority
                    (make-class-frobs nil (features new)
                                          value-var)))
(setf (rel-orient-var new) rel-orient-var)
; initialize parameter slots
(setf (dist-var new) dist-var)
(setf (rec-size new) rec-size)
(setf (boundary-list new) ; use corners as boundary list
      (get-frob-slot (features new) 'type 'corner))
(put-file (machine verifier) ; write boundary list to file
          clique-tester (mapcar #'car
                                (output {corner-finder})))
(clear verifier) ; clear hypothesis verifier output
(setf (args verifier) (conc "'-o' " clique-tester))
(pipe-and-run seg verifier) ; run hypothesis verifier
(setf (boundary-trans new) (output verifier))
(setf (cluster-size new) cluster-size)
(setf (local-clusters new) ; build and store model
      (build-lff-model (class-features new)
                       (features new) cluster-size))
(setf (model system-specs) new)))
```

```
;
; Recognition function
;
(defun lff-part-finder (system-specs)
  (let* ((model (model system-specs))
         (features (features system-specs))
         (classes (class-features model))
         (camera (camera system-specs))
         (segmenter (segmenter system-specs))
         (discrim (discriminator system-specs))
         (displayer (displayer system-specs))
         (verifier (verifier system-specs))
         (clique-tester (clique-tester system-specs))
         (shower (shower system-specs))
         (nodes nil) graph clique template x y
         perim-points (i-features nil))
    (clear camera)
    (operate camera)
    (clear segmenter)
    (pipe-and-run camera segmenter)
    (dolist (x features nil) ; run each feature
      (clear x) ; clear feature output
      (run-feature x system-specs) ; run feature
      (setf i-features (append i-features ; combine feature output
                               (make-feature-frobs x))))
    (assign-classes classes i-features) ; classify features
    (clear discrim) ; clear discriminator
    (setf (input discrim) (list clique-tester verifier segmenter
                                model i-features))
    (run discrim)
    (setf (args displayer) (conc "'-o' " (output discrim)))
    (clear displayer)
    (pipe-and-run camera displayer)
    (clear shower)
    (pipe displayer shower)))
```

C LFF Application Specific Rules

```
(def-rule LFF-set-cluster-size-accuracy
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
         (evalp (> (accuracy ?sys) (time ?sys))))
  :conc (progn
          (format -t "LFF-set-cluster-size-accuracy~%")
          (assert-val ?sys 'cluster-size 10)
          (assert-val ?sys 'rel-orient-var 0.30)
          (assert-val ?sys 'value-var 0.30)
          (assert-val ?sys 'dist-var 0.25)
          (assert-val ?sys 'rec-size 5)))

(def-rule LFF-set-cluster-size-rule
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
         (evalp (> (time ?sys) (accuracy ?sys))))
  :conc (progn
          (format t "LFF-set-cluster-size-time~%")
          (assert-val ?sys 'cluster-size 6)
          (assert-val ?sys 'rel-orient-var 0.30)
          (assert-val ?sys 'value-var 0.30)
          (assert-val ?sys 'dist-var 0.10)
          (assert-val ?sys 'rec-size 4)))

(def-rule LFF-select-modeler
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
         (evalp (undef (modeler ?sys))))
  :conc (progn (format t "LFF-select-modeler~%")
               (assert-val ?sys 'modeler 'build-lff-model)))

(def-rule LFF-select-segmenter
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
         (evalp (undef (segmenter ?sys))))
  :conc (progn
          (format t "LFF-select-segmenter~%")
          (assert-val ?sys 'segmenter ',{segmenter})))

(def-rule LFF-select-rle-converter
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
         (evalp (undef (rle-converter ?sys))))
  :conc (progn
          (format t "LFF-select-rle-converter~%")
          (assert-val ?sys 'rle-converter ',{rle-to-gray})))

(def-rule LFF-select-discriminator
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
         (evalp (undef (discriminator ?sys))))
  :conc (progn (format t "LFF-select-discriminator~%")
               (assert-val ?sys 'discriminator
                           ',{max-clique-finder})))

(def-rule LFF-select-displayer
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
         (evalp (undef (displayer ?sys))))
  :conc (progn (format t "LFF-select-displayer~%")
               (assert-val ?sys 'displayer ',{object-outliner})))

(def-rule LFF-select-shower
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
         (evalp (undef (shower ?sys))))
  :conc (progn (format t "LFF-select-shower~%")
               (assert-val ?sys 'shower ',{display-on-elan})))

(def-rule LFF-select-camera
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
```

```
        (evalp (undef (camera ?sys))))
  :conc (progn (format t "LFF-select-camera~%")
               (assert-val ?sys 'camera ',{ccd-camera})))

(def-rule LFF-select-features
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
         (evalp (undef (features ?sys))))
  :conc (progn (format t "LFF-select-features~%")
               (assert-val ?sys 'features
                           '(,{hole-finder} ,{corner-finder}))))

(def-rule LFF-select-ff-priority
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
         (evalp (undef (ff-priority ?sys))))
  :conc (progn (format t "LFF-select-ff-priority~%")
               (assert-val ?sys 'ff-priority
                           '((type hole) (type corner)))))

(def-rule LFF-select-clique-tester
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
         (evalp (undef (clique-tester ?sys))))
  :conc (progn (format t "LFF-select-clique-tester~%")
               (assert-val ?sys 'clique-tester "/tmp/ctemp")))

(def-rule LFF-select-verifier
  :type ((?sys system-specs))
  :prem ((method ?sys lff)
         (complete ?sys set)
         (evalp (undef (verifier ?sys))))
  :conc (progn (format t "LFF-select-verifier~%")
               (assert-val ?sys 'verifier ',{boundary-verifier})))
```

References

[1] S. Baase. *Computer Algorithms.* Addison Wesley, Massachusetts, 1978.

[2] H. Barrow and J. Tennenbaum. *Recovering Intrinsic Scene Characteristics from Images.* Technical Report 157, SRI International, April 1978.

[3] B.G. Batchelor, D.A. Hill, and Hodgson D.C. *Automated Visual Inspection.* Elsevier, New York, 1985.

[4] R.C. Bolles. *Recognition and Location of Partially Visable Objects.* Technical Report 11, SRI International, November 1982.

[5] R.C. Bolles. *Recognition and Location of Partially Visible Objects.* Technical Report 10, SRI International, November 1980.

[6] R.C. Bolles and R.A. Cain. Recognizing and Locating Partially Visible Objects: The Local-Feature-Focus Method. *Robotics Research*, 1(3):57–82, 1982.

[7] R.C. Bolles and P. Horaud. 3DPO: A Three-Dimensional Part Orientation System. *Robotics Research*, 5(3):3–26, 1986.

[8] L. Brownston, E. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5.* Addison Wesley, 1985.

[9] R.T. Chin and R. Dyer. Model-Based Recognition in Robot Vision. *Computing Surveys*, 18:67–108, March 1986.

[10] L.D. Erman, J.S. Lark, and F. Hayes-Roth. Engineering Intelligent Systems: Progress Report on ABE. In *Expert Systems Workshop*, Palo Alto, California, April 1986.

[11] J. Gaschnig, P. Klahr, H. Pople, E. Shortliffe, and A. Terry. *Building Expert Systems*, chapter 8, pages 253–272. Addison Wesley, Massachusetts, 1983.

[12] G.J. Gleason and Agin G.J. A Modular System for Sensor-Controlled Manipulation and Inspection. In *Proceedings of the 9th National International Symposium on Industrial Robots*, pages 94–104, Society of Manufacturing Engineers, March 1975.

[13] T.C. Henderson, C.D. Hansen, and B. Bhanu. The Specification of Distributed Sensing and Control. *Journal of Robotic Systems*, 2(4):387–396, 1985.

[14] C.C. Ho. *CAGD-based 3-D Object Representations for Computer Vision.* Master's thesis, University of Utah, Salt Lake City, Utah, June 1987.

[15] B.K.P. Horn. Obtaining Shape from Shading Information. In P. Winston, editor, *The Psychology of Computer Vision*, pages 115–155, McGraw-Hill, New York, 1970.

[16] P. Horn and B. Klaus. *Robot Vision.* Mcgraw-Hill, New York, 1986.

[17] *HP-RL Manual.* Hewlett-Packard, 1986.

[18] S.S. Vincent Hwang. *Evidence Accumulation for Spatial Reasoning in Aerial Image Understanding.* PhD thesis, University of Maryland, College Park, Maryland, December 1984.

[19] D. Kuan and R. Drazovich. Model-based Interpretation of Range Imagery. In *Proceedings of the National Conference On Artificial Intelligence*, pages 210–215, Mcgraw-Hill, 1983.

[20] D.M. Jr. Mckeown, W.A. Harvey, and Mcdermott. Rule Based Interpretation of Aerial Imagery. In *IEEE Workshop on Principles of Knowledge-Based Systems*, Denver, Colorado, December 1984.

[21] E Muehle. *FROBS Manual.* Technical Report PASS-note-86-11, University of Utah, October 1986.

[22] A.M. Nazif and M.D. Levine. Low Level Image Segmentation Expert System. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(5):555–577, September 1984.

[23] S. Shebs. *PCLS User Manual.* University of Utah PASS Group, 1985.