COPY

TR-88-3

Model-Based Vision System by Object-Oriented Programming

Huey Chang, Katsushi Ikeuchi, and Takeo Kanade

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by the Defense Advanced Research Projects Agency, DOD, through ARPA Order No. 4976, Amendment 20, and monitored by the Air Force Avionics Laboratory under contract F33615-87-C-1499 and under The Analytic Sciences Corporation sub contract 87123 modification No. 1.

Table of Contents

List of Figures

Abstract

This paper presents an approach to using object-oriented programming for the generation of a object recognition program that recognizes a complex 3-D object within a jumbled pile.

We generate a recognition program from an interpretation tree that classifies an object into an appropriate attitude group, which has a similar appearance. Each node of an interpretation tree represents a feature matching. We convert each feature extracting or matching operation into an individual processing entity, called an *object*. Two kinds of objects have been prepared: *data objects* and *event objects*. A data object is used for representing geometric objects (such as edges and regions) and extracting features from geometric objects. An event object is used for feature matching and attitude determination. A library of prototypical objects is prepared and an executable program is constructed by properly selecting and instantiating modules from it. The object-oriented programming paradigm provides modularity and extensibility.

This method has been applied to the generation of a recognition program for a toy wagon. The generated program has been tested with real scenes and has recognized the wagon in a pile.

1 Introduction

Traditionally, a recognition program is generated by a human expert who examines the features of an object, develops a strategy for a recognition procedure, and writes a specialized program for the individual object. However, this "hand writing" of a recognition program requires a long time for programming and testing. In order to reduce the development time, several researchers have investigated methods to automatically generate recognition programs from object models [5, 7, 8].

Automatic generation of a recognition program requires several key components:

- *object models* to describe the geometric and photometric properties of an object to be recognized;
- *sensor models* to predict object appearances from the object model under a given sensor;
- *strategy generation* using the predicted appearances to produce a recognition strategy;
- *program generation* converting the recognition strategy to executable program.

This paper concentrates on the final stage, i.e. program generation. We will investigate a way to automatically generate a program to localize an object under the assumption that its recognition strategy is given.

We propose to prepare a library of modules to be used for converting a strategy into a program and to construct the program by properly selecting modules from the library. Our method is based on object-oriented programming. An object in object-oriented programming is a processing unit, which can store several internal values in slots and execute various operations. This paper identifies the necessary operations in recognition strategies and prepares the prototypes of the objects to execute the strategies in the library. Then, this paper defines a generation method for an executable program by instantiating the objects in the library. Finally, this paper applies the method to a toy wagon to generate a recognition program and executes the generated program in a real scene to demonstrate the validity of our method.

2 Generating a Recognition Strategy

This section overviews our recognition strategy which is to be converted into an executable program in the following sections. Our paradigm is to generate a recognition program to localize a 3D object within a jumbled pile under the assumption that its geometric and photometric properties, sensor characteristics, and sensing conditions are known. The basic recognition strategy is to classify one unknown attitude (one object appearance) into one of several possible attitude groups by using various available features, and then to determine the precise attitude by solving equations based on the visible features of the group. Each group consists of topologically equivalent object appearances and is referred to as an aspect [9].

Strategy generation is performed by recursive sub-divisions of possible aspects by available features. Strategy generation starts with a root node which contains all possible aspects. After that time, whenever a new classification is done, new nodes are generated. At each node of the interpretation tree, each available feature is examined to determine whether it can classify the group of aspects in the node into a smaller number of aspects. If it can, the feature is stored at the node and subnodes corresponding to classified subgroups of aspects are generated and connected to the node. Thus, the generated recognition strategy is represented as a tree, which we call an interpretation tree. Intermediate nodes of the interpretation tree correspond to classification stages and leaf nodes correspond to classification into individual aspects [7].

Two kinds of features are used for matching: unitary features and relational features. A unitary feature can be represented as scalar numbers, such as area and moment of a visible face, while a relational feature is a detailed relational description between visible faces, such as face-face relations and face-edge relations.

At the completion of the aspect classification, each intermediate node of the interpretation tree records the feature to be used for classification, and each leaf node contains one single aspect. Suppose at this moment, we apply the interpretation tree to one object appearance[1]. Then, we can classify the appearance into the corresponding aspect at the leaf node by using the same features and values recorded at each intermediate node of the interpretation tree.

The next task will be to determine the exact attitude of the object within that aspect. Once an appearance is classified into an aspect, the interpretation tree knows the correspondence between image regions and object faces, in particular the correspondence between the entry region and the corresponding object face. Thus, once we define the local coordinates of the object face by the surface orientation of the face, the minimum moment direction of the face, and the

[1]More precisely, one image region of an object appearance is given to the interpretation tree. We will denote the image region from which the process begins as the entry region.

relationship between visible faces, we can recover the local coordinates relative to the world because those three piece of information can be obtained from the entry region. Then, the object attitude can be recovered from the local coordinates and the transformation from the local coordinates to the body coordinates of the object.

After the exact attitude of the object is obtained, the system generates an expected image by using a geometric modeler. Edges in the expected image will be compared with the edges in the input image to confirm the recognition. The voting index method provides a way to match the expected edges with the extracted edges by giving the reliability of the recognition. For the voting index see Appendix II.

3 Object Library

This section will consider how to convert a given strategy into an executable program. A recognition strategy is given as an interpretation tree in our system; each node of an interpretation tree contains a group of aspects and one of the feature matching operations to be used. We will identify necessary matching operations, and design *objects* to perform the operations by using the object-oriented programming technique.

An object in object-oriented programming is a processing unit, which can store several internal values in slots. We can define demon functions for each slot, where a demon function will be invoked implicitly when we retrieve a value from the slot or insert a value into the slot. An object can execute an operation explicitly when we send a particular message to the object. An object can be defined as an instance of a prototypical object. An instance object can inherit slot names, slot values, demon functions, and operations of the prototypical object[2]

Two kinds of objects are prepared in our object library. One is a *data object*, which is used in representing geometric objects (such as edge and region) and extracting features from geometric objects. The other is an *event object*, which is used to control the matching and determine the exact attitude after the interpretation.

[2]There are several implementations to the objects. In our system, we use modified Framekit+ originally developed at Carnegie Mellon University [2].

3.1 Data Object

Our system uses photometric stereo to obtain region information [6], and uses a line extractor to obtain edge information [1, 10]. To represent these pieces of information, we create two prototypical data objects in the object library. They are :

- Region
- Edge

The following example shows the definitions of the two abstract objects; an abstract-region and an abstract-edge.

```
(abstract-region-object
   (is-a program-object)
   (id-number)
   (area)
   (maximum-x)
   (minimum-x)
   (maximum-y)
   (minimum-y)
   (mass-center)
   (moment)
   (orientation)
   (region-search-distance)
   (region-image-model-distance-coef)
   (region-image-model-area-coef)
   (region-image-model-moment-coef)
   (region-area
      (if-needed-demon region-area-func))
   (region-moment
      (if-needed-demon region-moment-func))
   (region-moment-ratio
      (if-needed-demon region-moment-ration-func))
   (region-orientation
      (if-needed-demon region-orientation-func))
   (region-region-relation
      (if-needed-demon region-region-relation-func)))

(abstract-edge-object
   (is-a program-object)
   (id-number)
   (start-point)
   (end-point)
   (center)
   (length)
   (direction)
   (edge-region-relation
      (if-needed-demon edge-region-relation-func)))
```

In the definition of an abstract-region, the *is-a* slot represents that this abstract object is a program object. Slots from *id-number* through *orientation* will store image properties of individual regions by inheritance mechanism. Slots from *region-search-distance* through *region-image-model-moment-coef* keep global knowledge such as search distance for relational features or coefficients between data in the geometric modeler and image data. Slots from *region-area* through *region-region-relation* store features which are obtained from image properties by demon functions attached to the slots.

We can make instance objects of these abstract objects. When the instance objects are generated, the image properties of each region or edge are extracted from an image and stored in the corresponding slots. Thus, for example, an instance object of an region looks like;

```
(R10
   (instance abstract-region)
   (id-number 100)
   (maximum-x 100)
   (maximum-y 100)
   (minimum-x  50)
   (minimum-y  50)
   (mass-center (75 75))
   (moment      (8000 200 0.2))
   (orientation (0.0 0.0 1.0)))
```

The global knowledge and demon functions can be accessed from an instance object through the inheritance mechanism if necessary. For example, if the feature, region-area of the instance object, *R10* is accessed by a recognition process, there is no slot in *R10*. Thus, an inheritance mechanism is invoked and the region-area slot of the abstract-region is accessed. The demon function attached to the region-area slot of the abstract-region is invoked. Then, the demon function calculates the region-area of *R10* by using *region-image-model-area-coefficient* in the abstract region and the area value in R10 and returns the feature value to the recognition process.

This mechanism makes the access format of the image features (say, region-area) by the recognition process independent of the output format of image properties (say, image area) given by a sensor. In particular, this mechanism is convenient when we handle multiple sensors. Each sensor has a particular output format and model-image coefficients. Thus, if we use the conventional method without demon functions, we have to exchange access functions of the recognition process depending on sensors and features. However, if we use this demon mechanism, we not need to change the access functions of the recognition process; we only need to redefine demon functions. Since the global knowledges and all the demon functions are attached only to the abstract region and the abstract edge, necessary changes are localized at the level of the abstract region and abstract edge.

The relational features such as region-region or region-edge are also represented by using demon functions. These relational features are represented relatively with respect to each region. If we use the conventional method, we have to calculate all relational features with respect to all regions beforehand, even though most of them are unnecessary. Since the calculation of a relational feature is expensive, it is desirable to reduce the amount of calculation by using demon functions which calculate those features only when they are actually required.

3.2 Event Object

Event objects are used to convert nodes of an interpretation tree into executable modules for feature matching and attitude determination. There are two kinds of features to be used for matching; unitary features such as area or moment and relational features such as region-region relation or region-edge relation. We convert a node for a unitary feature into an object which chooses one of the descendant nodes simply based on the value of the unitary feature of a region. On the other hand, we will convert a node for a relational feature into an object which examines the similarity of the relational feature to all possible cases and determines the node corresponding to the most likely case.

3.2.1 Unitary feature object

When a node of an interpretation tree is required to examine a unitary feature, an unitary feature object is generated and attached to the node. A node of an interpretation tree contains the information about descendant nodes, the name of unitary feature used for matching, and its threshold value. According to these pieces of information, a unitary feature object is generated. Thus, the prototype of a unitary feature object in the object library has the following format.

```
(unitary-feature-object
   (is-a program-object)
   (execution)
   (threshold)
   (branch-left)
   (branch-right))
```

When an instance unitary feature object is generated, it contains a method name in the *execution* slot to be used for the comparison, the threshold value in the *threshold* slot. For example, if an interpretation tree requires area comparison at a particular node, then the following object will be generated at the node.

```
(branch-example-1
   (instance unitary-feature-object)
   (execution area-comparison-method)
   (threshold 100)
   (branch-left branch-example-10)
   (branch-right branch-example-11))
```

The threshold value, branch-left, branch-right, and the execution method name are obtained from the interpretation tree and inserted by this conversion process. The object library contains the following function.

```
(defun area-comparison-method (schema slot entry-region)
 (cond((unitary-comparison
          (get-value entry-region 'region-area)
          (get-value schema 'threshold))
        (send (get-value schema 'branch-left)
              'execution entry-region))
      (t(send (get-value schema 'branch-right)
              'execution entry-region))))

(defun unitary-comparison(arg-a arg-b)
  (cond((>= arg-a arg-b) t)(t nil))))
```

The *area-comparison-method* is invoked by sending an execution message to the object such as

```
 (send 'branch-example-1 'execution entry-region).
```

In the arguments of the method function, *schema* and *slot* are the corresponding schema and slot which invoke this function and inserted by the system; in our example, *branch-example-1* and *execution* are inserted automatically, while the argument, entry-region is given to this method function directly by the *send* function[3]. Depending on the result from unitary-comparison, another execution message will be sent either to *branch-example-10* or *branch-example-11*.

Similarly, we can define various discrimination functions, where required functions are dependent on the strategy generation. In the present implementation, the following functions are prepared in the object library;

---

[3]Note that *(get-value entry-region 'region-area)* invokes a region-area demon function attached to the abstract-region.

- area-comparison-method,
- moment-comparison-method,
- moment-ratio-comparison-method,
- surface-characteristic-comparison-method,
- surrounding-nth-face-area-comparison-method,
- surrounding-nth-face-moment-comparison-method,
- surrounding-nth-face-moment-ratio-comparison-method,
- surrounding-nth-face-surface-characteristics-comparison-method.

It is quite easy to include different unitary features. This only requires addition of the necessary feature matching methods and the feature slot with the feature extraction demon to the library; it is not necessary to modify any other existing objects.

3.2.2 Relational feature object

If a node of an interpretation tree is required to examine a relational feature, a parallel tracking mechanism is adopted which examines the similarity of the relational features of all immediate descendant nodes against those of the entry region and sends the next execution message to the node corresponding to the highest similarity.

Since the parallel tracking mechanism is relatively complicated, we divide it into the following four kinds of objects; a message handling object, feature matching objects, feature matching demon objects, and a comparing object. See Figure 1. A message handling object sends execution messages to feature matching objects. A feature matching object measures a similarity between the feature of the entry region and one of the model features with the help of a feature matching demon object, and then sends the similarity measure to the comparing object, and a finish notice message to the message handling object. Once the message handling object receives all finish notice messages from all feature matching objects, it invokes the comparing object. The comparing object examines the similarity measures and sends the next execution message to the appropriate object.

Message handling object

The message handling object controls the parallel matching mechanism. It sends the model

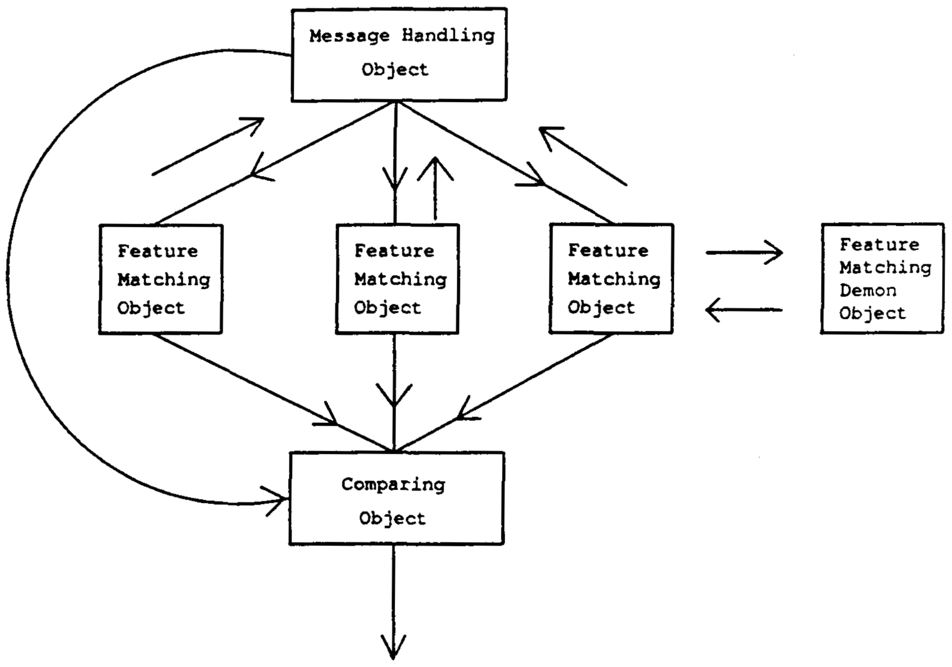

Figure 1: Parallel tracking mechanism

features to each feature matching object one by one. The prototype of the message handling object has the following format.

```
(message-handling-object
   (is-a program-object)
   (execution message-handling-method)
   (finished-notice finished-notice-method)
   (sending-object-list)
   (finished-object-list)
   (model-feature-list)
   (next-node-list)
   (comparing-object)
```

The slot, *model-feature-list* contains the model relational features given from the node of the interpretation tree. The slot, *sending-object-list* contains the feature matching objects, where those feature matching object will be generated while the system converts the interpretation tree into an executable code and registers them in this slot, while the slot, *finished-object-list* contains the feature matching object which finishes the matching operation and sends the notice to this object. Once all model matching is done, a comparing object is invoked. The object to be invoked is stored in the *comparing-object* slot.

The object library contains the following *message-handling-method* and *finished-notice-method.*

```
(defun message-handling-method(schema slot entry-region)
 (do((model-list (get-value schema 'model-feature-list)
                 (cdr model-list))
     (sending-list
                 (get-value schema 'sending-object-list)
                 (cdr sending-list))
     (node-list  (get-value schema 'next-node-list
                 (cdr node-list)))
    ((null model-list))
  (send (car sending-list)  'execution
        entry-region (car model-list)
                     (car node-list))))
```

Basically, this method sends model relational features one by one to feature matching objects. In order to make a correspondence between a feature and the corresponding descendant node, this method also send the names of the descendant nodes to the feature matching objects.

```
(defun finished-notice-method
              (schema slot sender entry-region)
 (add-value schema 'finished-object-list sender)
 (cond((=
       (length(get-values schema 'finished-object-list))
       (length(get-values schema 'sending-object-list)))
      (send (get-value schema 'comparing-object)
            'execution entry-region))
```

This method adds the senders name in the *finished-object-list* everytime it receives a finished notice from a feature-matching object. If all the feature matching objects, invoked by this object, finish their matching operations, the message handling object sends an execution message to the comparing object.

Feature matching object

The feature matching object performs the relational feature matching. The prototype of the feature matching object has the following format.

```
(feature-matching-object
   (is-a program-object)
   (execution feature-matching-method)
   (finished-notice finished-notice-method)
   (comparing-object)
   (message-handling-object)
   (feature-matching-demon-object)
   (node))
```

Those *comparing-object*, *message-handling-object*, and *feature-matching-demon-object* contain object names corresponding to those slot names and are filled by the conversion process. The slot, *feature-matching-method* contains an execution method to examine the similarity between the feature sent by the message handler and those of the entry region, while the main body of the calculation is done by *feature-matching-demon-object.* These methods can be represented in the library as

```
(defun feature-matching-method
  (schema slot entry-region model-feature node)
  (new-value schema 'node node)
  (send (get-value schema 'feature-matching-demon-object)
        'execution entry-region model-feature)))

(defun finished-notice-method
   (schema slot score)
   (send (get-value schema 'comparing-object)
         'add-value
         (get-value schema 'node)
          score)))
```

Feature matching demon object

The feature matching demon object measures the similarity between the model-feature and features of the entry-region. This function further invokes demon functions attached to the entry region to get either region-region relations or region-edge relations and, then, calculates the similarity measure between them by using a similarity measuring method. The resulting measure will be returned to the feature matching object and then sent to the comparing object. The prototypical object in the library has the following format;

```
(feature-matching-demon-object
   (is-a program-object)
   (execution)
   (feature-matching-object))
```

The slot, *feature-matching-object* contains the object name which invokes this object. This will be done by the conversion process. The slot, *execution* contains a similarity measuring method. In the present implementation, the following two methods are prepared in the library.

- region-region similarity measuring method
- region-edge similarity measuring method

Similarity of the region-region relational feature and the region-edge relational feature are measured based on the voting index. For relational features, see Appendix I, and for voting indices see Appendix II. If a different similarity measure is necessary, it is only necessary to add the method to the library and to insert the method name into the *execution* slot of this object.

Comparing object

Each time a comparing object receives a message *add-score* with the similarity measure and the node from a feature matching object, it will add the measure to the score list and the node to the next node list. After the message handling object finishes its sending to the feature matching objects, it sends an execution message to a comparing object and invokes it. The comparing object examines the similarity measures in slot "score-list", chooses the highest measure, and sends the next execution message to the node corresponding to the highest measure. Thus, the prototype of the comparing object has the following format.

```
(compare-object
  (is-a program-object)
  (execution compare-object-method)
  (add-score add-score-method)
  (score-list)
  (next-node-list))
```

The following two methods are also prepared in the library.

```
(defun compare-object-method (schema slot entry-region)
   (send (the-most-highest-node
            (get-value schema 'score-list)
            (get-value schema 'next-node-list))
            'execution entry-region))

(defun add-score-method(schema slot score node)
  (add-value schema 'score-list score)
  (add-value schema 'next-node-list node))
```

where the function *the-most-highest-node* returns the node in the next-node-list which has the highest value in the score list.

3.2.3 Attitude determination object

An attitude determination object is generated at a leaf node of an interpretation tree. At each leaf node, the interpretation tree knows the correspondence between the image regions and model faces, in particular one between the entry region and the corresponding model face. If we recover the local coordinate of the model face from the information of the entry region, then we can obtain the body coordinate by using the local coordinate and the transformation from the local coordinate to the body coordinate obtained from the geometric model. In our system, we define the local z axis by the surface orientation, x axis by the minimum moment direction and visible face relationships. Once this object determines the body coordinate, it sends the coordinate to the verification object.

The prototypical attitude determination object has the following format.

```
(attitude-determination-object
   (is-a program-object)
   (execution attitude-determination-method)
   (transformation)
   (verification-object))
```

3.2.4 Verification object

The verification object is used to generate an expected image and verify the recognition result. After the exact attitude is determined, the verification object will create an expected image by using a geometric modeler. From the expected image, it will extract 2D edge informations and match this with the input scene to confirm the recognition.

```
(verification-object
   (is-a program-object)
   (execution verification-method))
```

4 Generating an Executable Code for a Toy Wagon

We choose a toy wagon to demonstrate our ideas. We use a geometric modeler to generate a model of the toy wagon. Figure 2 shows the model of the toy wagon. It is a relatively complex geometric object. In order to derive possible aspects, we sample possible views and group them into 17 aspects based on the visible faces. Figure 3 shows the given interpretation tree, which defines the necessary feature matchings at each node.

Once the interpretation tree is obtained, its nodes are converted to objects using the object library.

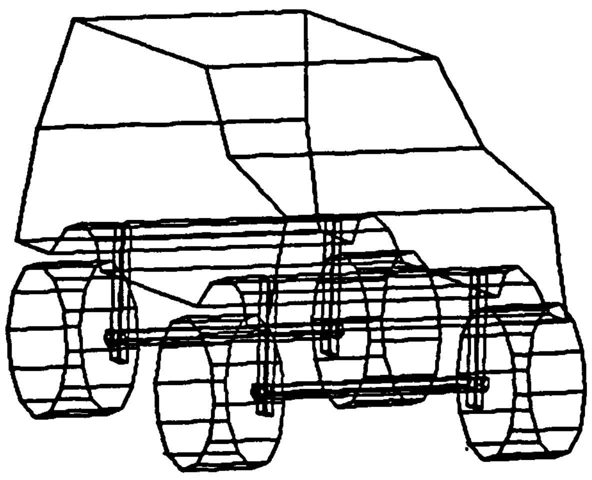

Figure 2: The model of a toy wagon

At the nodes, *b1, b11, b111* of the interpretation tree, one unitary feature matching node is converted into one unitary feature matching object.

For example, at node b1 of the interpretation tree, the following object is generated.

```
(b1
  (execution moment-comparison-method)
  (threshold 5000)
  (branch-left b11)
  (branch-right b12))
```

where threshold value 5000 is given from the interpretation tree. A similar object is generated at *b11, b111* by using the same moment feature and different threshold value.

A relational feature feature is matched using a parallel tracking mechanism. A parallel tracking mechanism is divided into four objects; message handling object, feature matching object, feature matching demon object, and compare objects. These objects are generated when a parallel tracking mechanism is required by the conversion program.

Those nodes *b12,b112,b122,b121,b1111,b1112,b1121,b11121* require relational feature matching, and thus, are converted into objects to execute the parallel tracking mechanism. Let

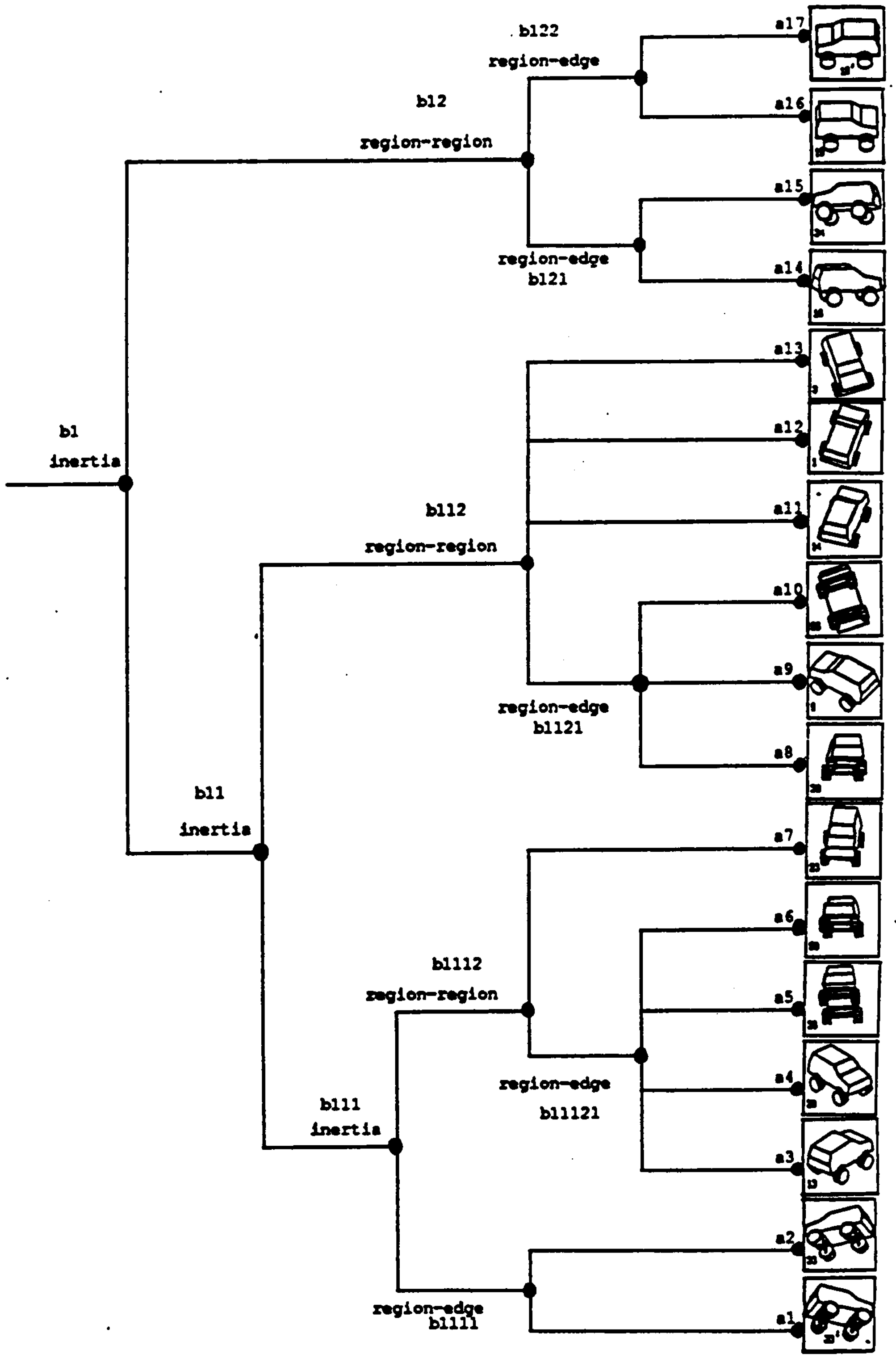

Figure 3: Interpretation tree for a toy wagon

us consider the case of *b112*, at which node a region-region relational feature is used in matching. The conversion program instantiate one message handling object *b112*, four feature matching objects, *b112-f-1,..,b112-f-4* four feature matching demon objects, *b112-f-1-d,..,b112-f-4-d* and one comparing object, *b112-c* from those prototypical objects in the library.

First, a message handling object such as

```
(b112
   (instance 'message-handling-object)
   (sending-object-list
      '(b112-f-1 b112-f-2
        b112-f-3 b112-f-4)
   (finished-object-list nil)
   (model-feature-list
       '(((10 20 30 0.5)) ...))
   (next-node-list
       '(a13 a12 a11 b1121))
   (compare-object b112-c))
```

is generated. The contents in the *model-feature-list* slot is obtained from the relationship between the entry region and surrounding visible regions consulting a model data base, and represent region-region relational features such as the distance between regions and the difference between two surface normals. More precise definitions can be found in Appendix I region-region feature.

Then, four feature matching objects are instantiated from the prototype in the object library. One of them looks like this:

```
(b112-f-1
   (instance feature-matching-object)
   (comparing-object b112-c)
   (message-handler-object b112)
   (feature-matching-demon-object
                    b112-f-1-d))
```

Then four feature-matching-demon-objects, instantiated from the prototypical object in the library, have the same format as the feature-matching-objects.

Then, finally, a comparing object is instantiated.

```
(b112-f-c
   (instance comparing-object)
   (score-list nil)
   (next-node-list nil))
```

At each leaf node, attitude determination objects and verification objects are generated. For example, at node a9, the following two objects are generated.

```
(a9
   (instance attitude-determination-object)
   (transformation
     ((0.0 0.0 1.0) ...)))
   (verification-object a9-v))

(a9-v
   (instance verification-object))
```

Note that some of the instance objects do not have execution slots, which are inherited from their prototypes in the object library.

Similar operations are applied to all nodes in the interpretation tree and give the executable program as shown in Figure 4. This conversion program is implemented using a rule representation language OPS5 [4].

5 Running the Code

This section shows an example of the obtained program running on a real scene. Figure 5 is the input scene for recognition. Figure 7 shows those regions whose surface orientation can be determined as shown in Figure 6 by using photometric stereo. By using a dual photometric stereo system, we can determine the depth of each region. We also use an edge extractor. Three images obtained under different lighting conditions are processed. The resulting edges are shown in Figure 8. The system instantiates region objects and edge objects for all the regions and edges in the scene by using the abstract-region object and the abstract-edge object in the object library.

The largest region at the top of the pile is selected as the entry region (in this case, region *r90* in Figure 5(c)) and sent to b1.

```
(send 'b1 'execution entry-region)
```

where *entry-region* = *R90*. Then, since b1's execution slot contains the moment-comparison-method, the moment comparison method is invoked. This function sends a message to region *R90* to get region-moment, which can be calculated by the region-moment demon function and the moment value of *R90*. Notice here that the moment in an image is converted into a moment value in the geometric model by the demon function. See Figure 9.

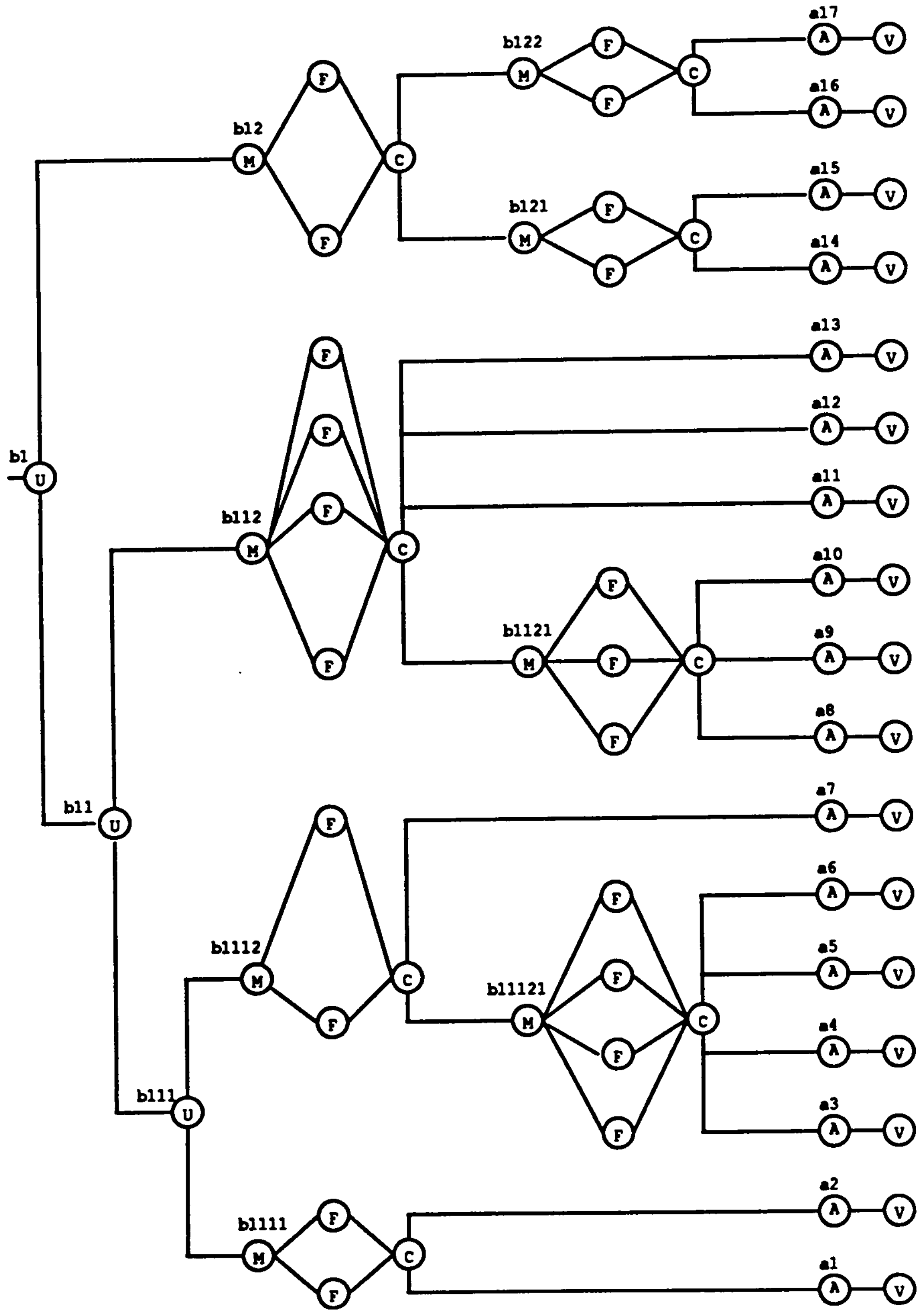

Figure 4: An executable program represented by objects: A U node represents an unitary feature matching object; A M node represents a message handling object; A F node represents a relational feature matching object; A C node represents a comparing object; An A node represents a attitude determining object; A V node represents a verification object.

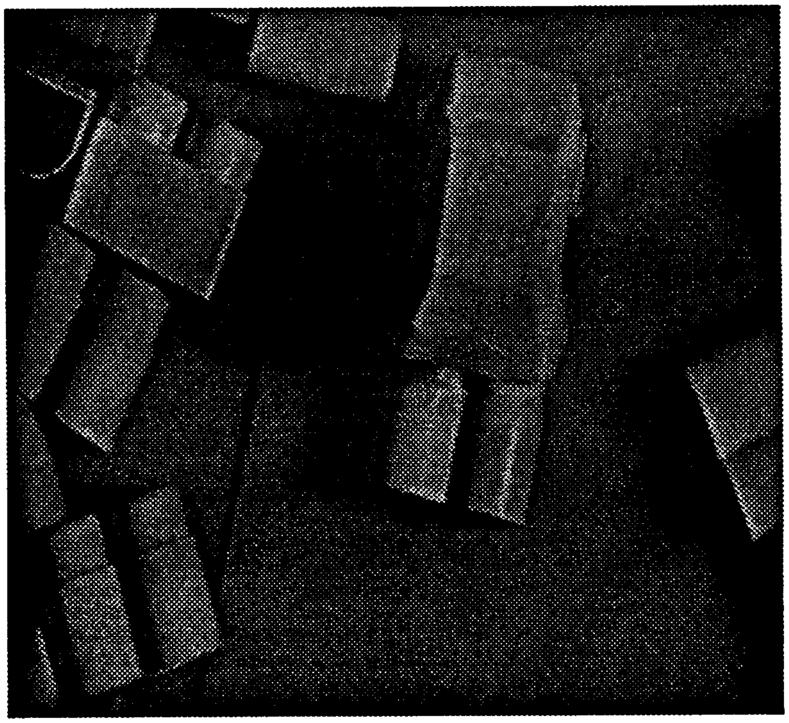

Figure 5: Input scene for the recognition

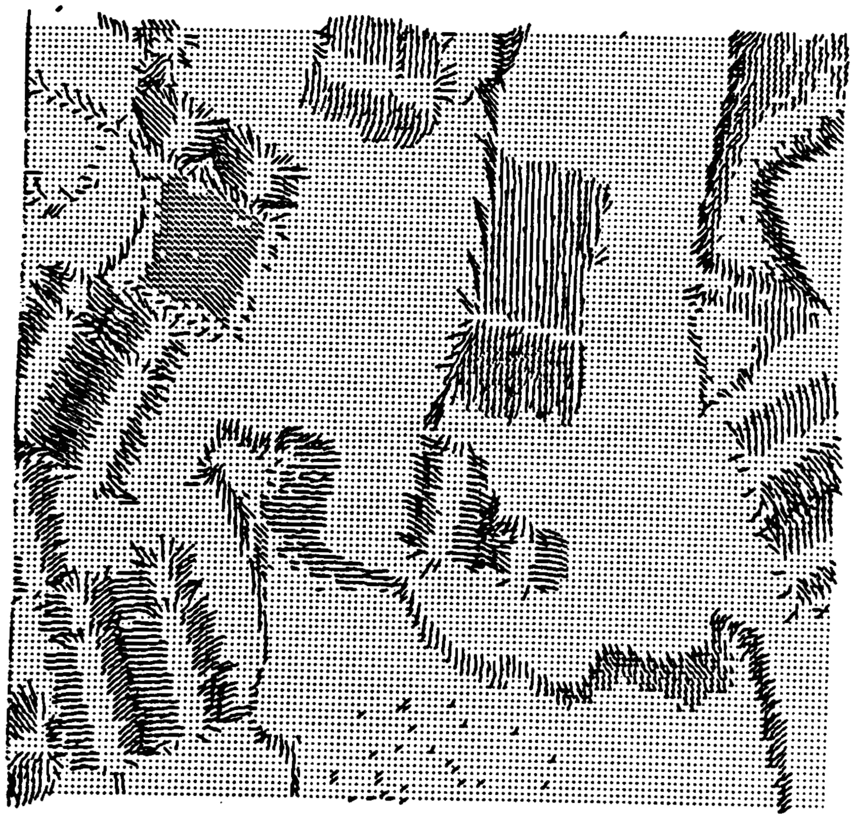

Figure 6: Needle map obtained by Photometric Stereo

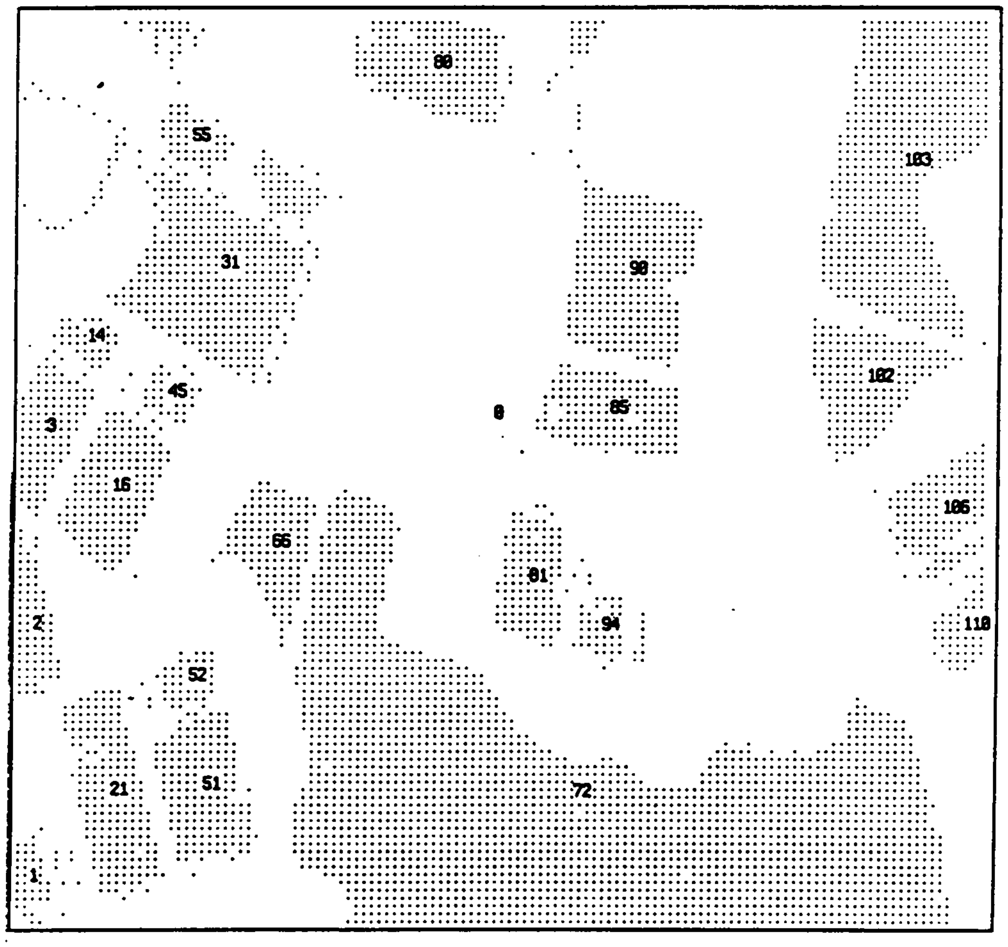

Figure 7: Regions obtained from the needle map

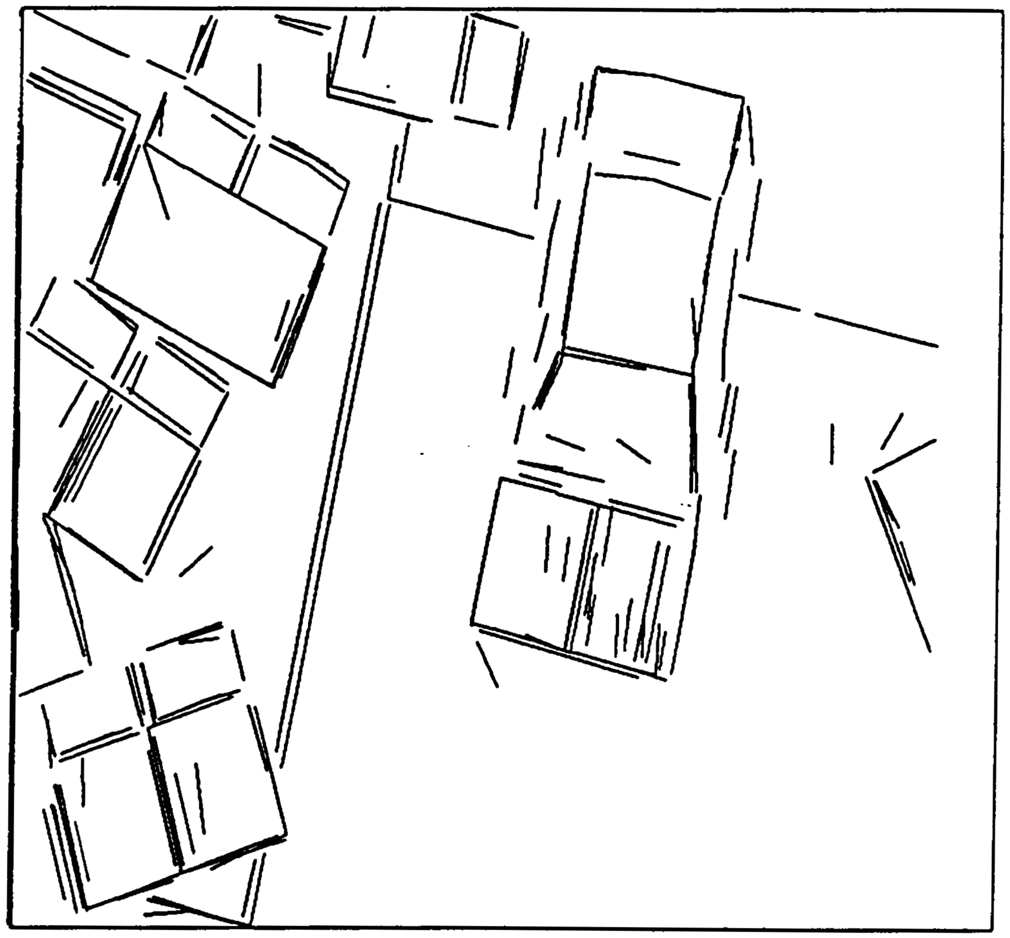

Figure 8: Edges obtained by Miwa Line Finder

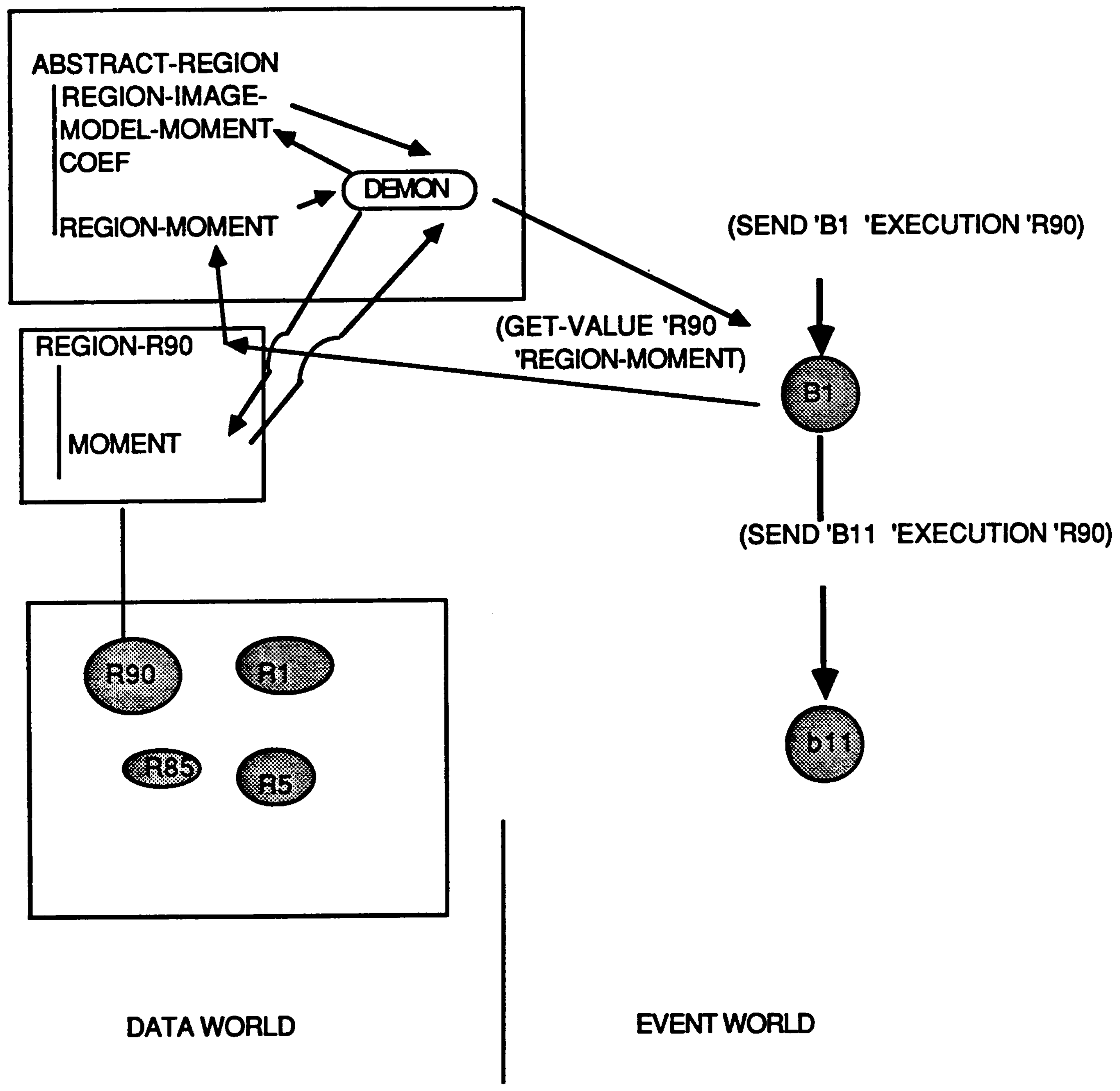

Figure 9: Retrieving feature value from a region.

From the comparison between the threshold value in the unitary feature matching object, *b1* and the feature value obtained from *R90*, the object sends an execution message and the entry region to *b11*. The object *b11* repeats the similar operation and sends an execution message and the entry region to *b112*. Since *b112* is a message handling object, it send messages *b112-f-1, b112-f-2,b112-f-3,b112-f-4*, one by one with model relational features. At each feature matching object, a similarity measure for the region-region relational feature obtained from the region-region relationship (*R90* and *R85*) against one model relational feature, is obtained and sent to the comparing object, *b112-c*. From the accumulated score, the comparing object, *b112-c* send an execution message to node *a11* with the entry region.

At this point, the system finds the correspondence between the entry region and the roof face of the toy wagon. The attitude determination object then determines the local coordinates of the face by using the surface normal, the minimum moment direction, and the region-region relation between *R90* and *R85*. The bold lines in Figure 10 indicate the tracks of the message passings. Finally, the body coordinates are recoverted using the local coordinates and the transformation between the roof face of the toy wagon and the body coordinates. The attitude determination object, *a11* sends an execution message to *a11-v* with the entry region and the body coordinates.

The verification object, *a11-v* generates an expected image (Figure 11) by using a geometric modeler based on the body coordinates, extracts edges from the expected image which are longer than a certain threshold, and compares them with the edges from the line finder. The result is shown in Figure 10(c), where the bold lines indicate the expected edges and thin lines indicate the image edges. The voting index obtained from this matching represents the reliability of the recognition. For this example, the reliability of the recognition is 0.8.

6 Conclusion

This paper has discussed how various modules are prepared and used for generating a recognition program from a given interpretation tree so that we can generate a recognition program from a geometric model automatically. We designed the module set as the object library using object-oriented programming. The object-oriented programming paradigm provides modularity and extensibility to the object library. The objects in the object library are divided into two categories: data objects and event objects. A data object is used for representing geometric objects and extracting features from geometric objects. An event object is used for

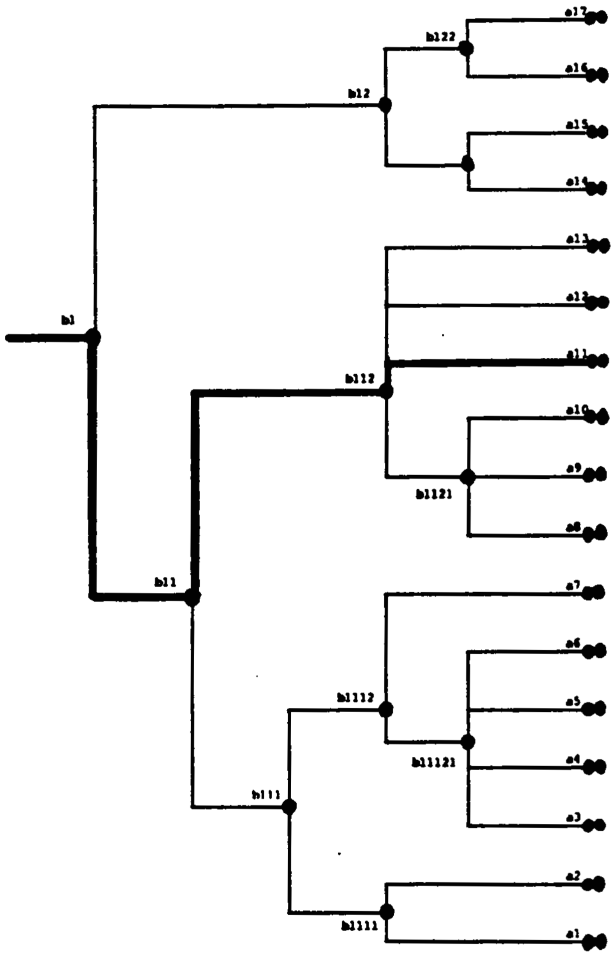

Figure 10: Execution of the program

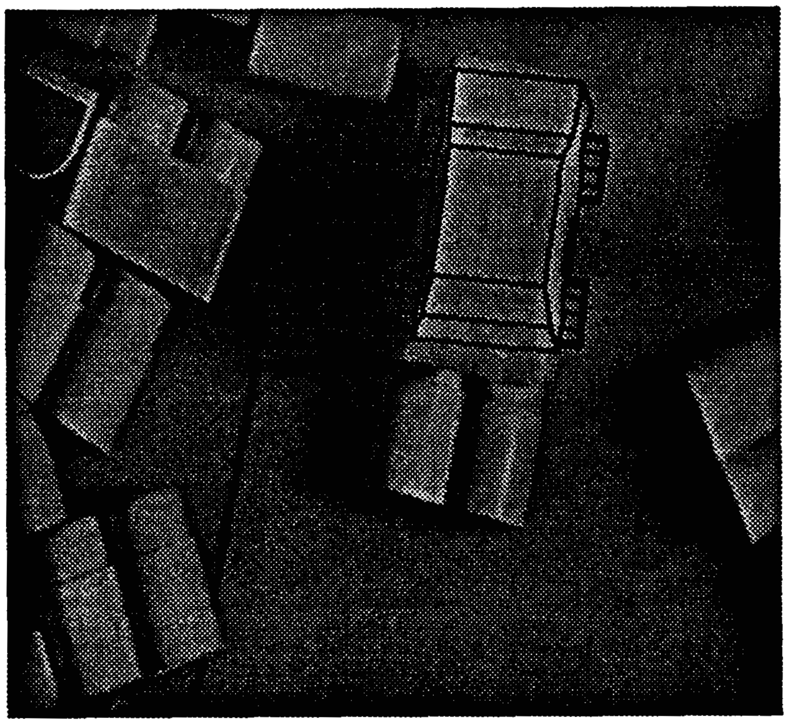

Figure 11: Superimposed image of the geometric model onto the image

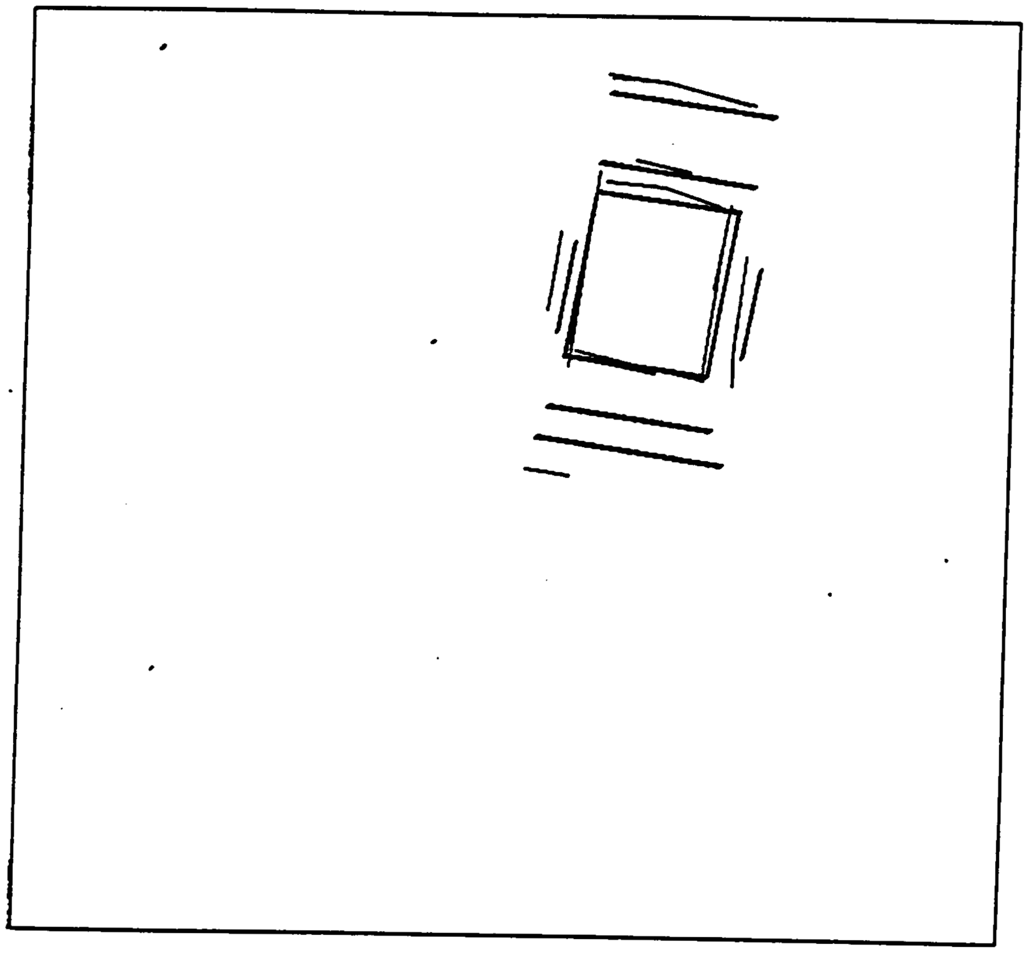

Figure 12: Verification by the extracted edges

feature matching and attitude determination. We generate an executable program by properly selecting and instantiating modules from the object library. This method has been applied to the generation of a recognition program for a toy wagon. The generated program has been tested with real scenes and has recognized the wagon in a pile. The generation method developed here provides a useful tool for the automatic generation of recognition programs.

Acknowledge

The authors thank Keith Gremban, Yoshinori Kuno, and the member of VASC (Vision and Autonomous System Center) of Carnegie Mellon University for their valuable comments and discussion.

I. Relational Features

I.1 Region-region Relational Feature

The relationship between two regions can be described as (Figure I-1):

- d : The distance between the mass centers of two regions.
- α : The angle between the minimum moment directions of two regions.
- β : The angle between the surface orientations of two regions.
- A : The area of the region other than the entry region.

We form a four-dimensional feature vector (d α β A) to represent the relational feature between the entry region and the other region. A demon function will be invoked when feature extraction is requested. Then a set of feature vectors relative to the entry region are found. These feature vectors will be used in feature matching.

In the four-dimensional feature space (d α β A), we test a hypothesis by comparing all the feature vectors with the predicted feature vector that is generated by the model. If they are close in the four-dimensional feature space, we accept this hypothesis, and conclude the feature matching process. If the matching fails, we then reject the hypothesis, and generate another hypothesis. This hypothesis generation and test can be done by using a parallel tracking schema.

I.2 Region-edge Relational Feature

We use a line finder to obtain 2-D information about an edge from its projection onto the image plane. In order to recover the 3-D information about an edge, we will transform the 2-D edge into 3-D space via an affine transformation. Let the surface orientation of the entry region be (p q), where p= u_x/u_z, and q= u_y/u_z. The affine transformation P will transform an edge surrounding the entry region to the 3-D plane that the edge lies on.

$$P = \begin{vmatrix} \sqrt{1+p^2} & pq/\sqrt{1+p^2} \\ 0 & \sqrt{1+p^2+q^2}/\sqrt{1+p^2} \\ 0 & 0 \end{vmatrix}$$

Figure I-2 shows the X-view of an affine transformation. The new view direction is on the Z' axis. We can determine the original length of an edge from this view direction.

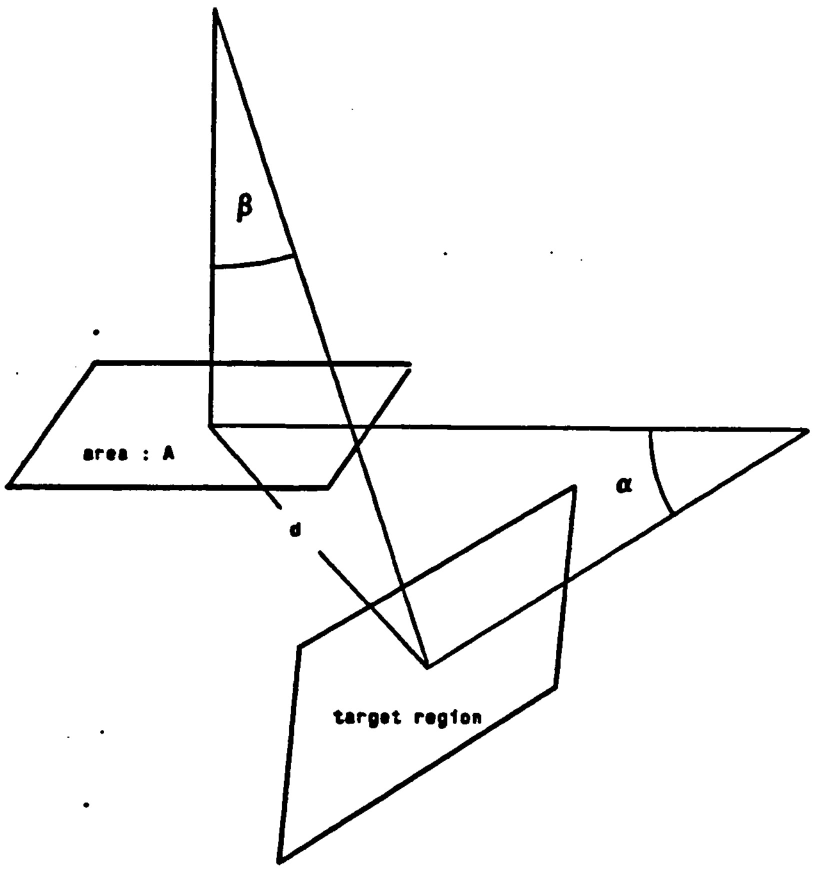

Figure I-1: Relation Between Two Regions

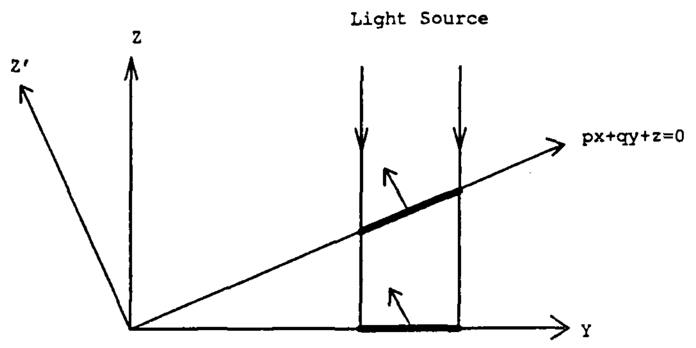

Figure I-2: X-view of affine transform

After the affine transformation, we can use four parameters to describe the relationship between an edge and the entry region (Figure I-3).

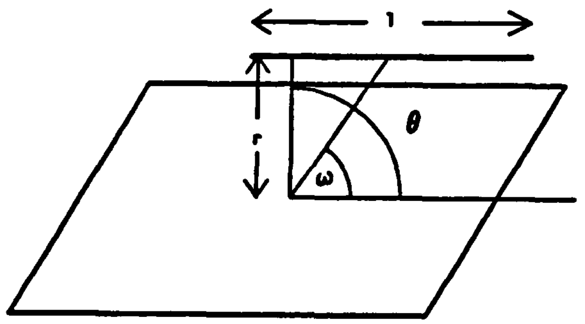

Figure I-3: Relation between edge and region

- r : The perpendicular distance between an edge and a region.

- θ : The angle from the minimum moment direction of a region to the perpendicular line of an edge.
- ω : The angle from the minimum moment direction of a region to the middle line of an edge.
- l : The length of the edge after an affine transformation.

For all the edges within the search distance, we generate feature vectors relative to the entry region. For each model edge, we search the feature vectors in the scene to find the voting index [3]. The summation of the voting index is compared with the total length of the model edges that surround the entry region. If the values are close, then we conclude this matching is successful, otherwise we reject this hypothesis and generate another hypothesis.

II. Voting Index

We generate region-edge relational features for the edges within a certain distance from the entry region. These features will be compared with the model's region-edge features generated in advance. The length of an edge in the scene is considered as a *vote* for the presence of a model edge if the following conditions are satisfied :

- The value r of the edge is within a certain range of the model edge r_m, or say,
  $0.9\,r_m < r < 1.1\,r_m$.
- The value θ of the edge is within a certain range of the model edge θ_m, say,
  $-0.2+\theta_m < \theta < 0.2+\theta_m$.
- The value of ω of the edge is within the maximum and minimum values of the model edge ω_m :
  $$\omega_m - \frac{l_m/2 + r_m \tan(\omega_m - \theta_m)}{r_m} < \omega < \omega_m + \frac{l_m/2 - r_m \tan(\omega_m - \theta_m)}{r_m}$$
- The length l of the edge is less than the length l_m of the model edge,
  $l < l_m$.

References

[1] Canny, J. F.
Finding edges and lines in images.
Technical Report AI-TR-720, Artificial Intelligence Laboratory, M. I. T., 1983.

[2] Carbonell, J. and Joseph, R.
Framekit+:A knowledge representation system.
Technical Report CS-TR, Computer Science Department, Carnegie Mellon University, March, 1986.

[3] Chang, H.
A Vision Algorithm Generator by Object-Oriented Programming.
Technical Report, Department of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburgh, PA, July, 1987.

[4] Forgy, C.L.
OPS5 User's manual.
Technical Report CMU-CS-81-135, Computer Science Department, Carnegie Mellon University, july, 1981.

[5] Goad, C.
Special purpose automatic programming for 3D model-based vision.
In *Proc. Image Understanding Workshop.* DARPA, 1983.

[6] Ikeuchi, K.
Determining a depth map using a dual photometric stereo.
The International Journal of Robotics Research 6(1), 1987.

[7] Ikeuchi, K.
Generating an Interpretation Tree from a CAD Model for 3-D Object Recognition in Bin-Picking Tasks.
International Journal of Computer Vision 1(2), 1987.

[8] Ikeuchi, K. and Kanade, T.
Towards automatic generation of object recognition program.
Proc. of IEEE (11), November, 1988.

[9] Koenderink, J. J. and Van Doorn, A. J.
Geometry of binocular vision and a model for stereopsis.
Biological Cybernetics 21(1), 1976.

[10] Miwa, H. and Kanade, T.
Line extraction.
Internal Memo., in preparation, Carnegie-Mellon University, Pittsburgh, PA, 1987.