A Versatile Computer-Controlled Assembly System.

Conference Paper · January 1973

Source: DBLP

CITATIONS
102

READS
116

5 authors, including:

Harry Barrow

34 PUBLICATIONS **3,655** CITATIONS

SEE PROFILE

Christopher M. Brown

University of Rochester

131 PUBLICATIONS **7,663** CITATIONS

SEE PROFILE

A Versatile System for Computer-Controlled Assembly[1]

A. P. Ambler, H. G. Barrow,[2] C. M. Brown, R. M. Burstall and R. J. Popplestone

Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland

Recommended by T. Sakai

ABSTRACT

A versatile assembly system, using TV cameras and computer-controlled arm and moving table, is described. It makes simple assemblies such as a peg and rings and a toy car. It separates parts from a heap, recognizing them with an overhead camera, then assembles them by feel. It can be instructed to perform a new task with different parts by spending an hour or two showing it the parts and a day programming the assembly manipulations. A hierarchical description of parts, views, outlines, etc. is used to construct models, and a structure matching algorithm is used in recognition.

1. Introduction

A computer-controlled versatile assembly system has been programmed using the Edinburgh hand-eye hardware (Barrow and Crawford [3]; Salter [14]). The equipment (Fig. 1) consists of a movable table, a mechanical hand with sensors and rotating palms and two TV cameras, all connected via an 8K Honeywell 316 to a 128K time-shared ICL 4130 running POP-2 programs. Several other programs are running on this equipment, including a program for recognizing irregular objects and one which packs arbitrarily shaped objects into a box (Michie et al. [12]). The program described here is our most ambitious effort. It is capable of assembling a variety of structures, and much of our effort has been spent in enabling the machine to acquire descriptions of the parts for itself using an overhead TV camera.

[1] The work was carried out when the authors were members of the now defunct Department of Machine Intelligence and briefly reported in the 1973 IJCAI conference.

[2] Present address: Stanford Research Institute, Stanford, California, U.S.A.

Artificial Intelligence 6 (1975), 129–156

Related work has been carried out at Hitachi (Ejiri et al. [8]), at MIT (Winston [17, 18]) and at Stanford University (Feldman [9]). The Hitachi program could build a variety of simple structures of blocks from line drawings of the structure, the MIT programs can learn concepts about structures and copy an arbitrary structure of simple blocks given spare parts, and a recent Stanford program can assemble a simple automobile water pump using pre-programmed hand manipulations.

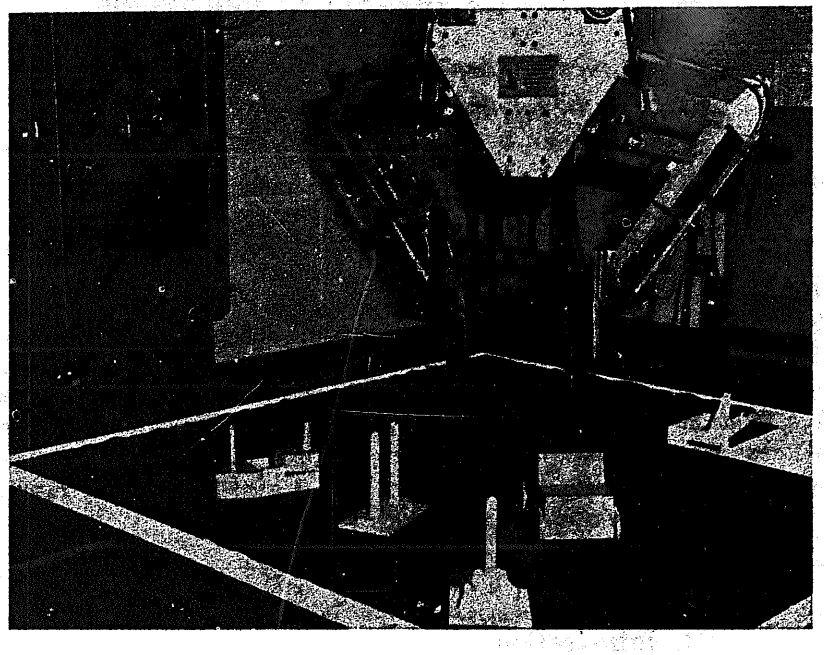

FIG. 1. Overview of the equipment.

2. The Task

2.1. Specification

A number of parts are placed by the operator in a heap on the table (Fig. 2, peg and rings). The machine's task is to separate the parts and recognize them, then to assemble them into some predetermined configuration (Fig. 3). Figs. 4 and 5 show another example, a toy car. We are thinking in terms of up to a dozen parts with outlines described by up to twenty or so straight or curved segments from any one view, possibly with some holes of similar complexity.

In order to explore the capabilities of a computer-controlled system as opposed to a conventional electromechanical device we seek a *versatile* assembly system. The demand for versatility is also calculated to raise interesting aspects from an Artificial Intelligence point of view.

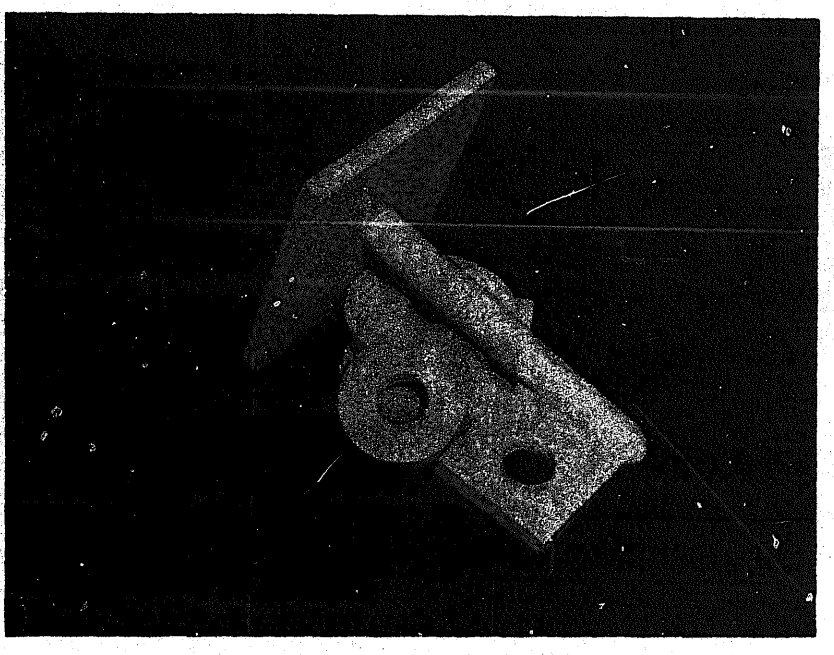

FIG. 2. Parts for the peg and rings.

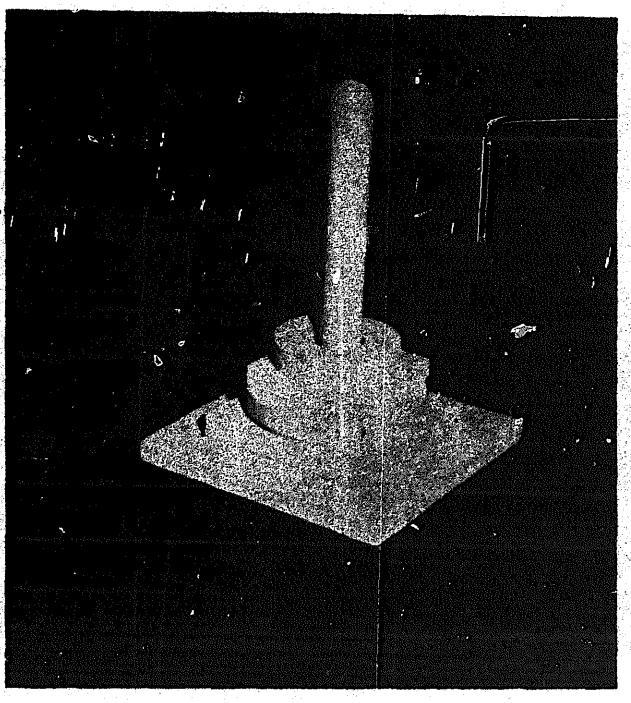

FIG. 3. The peg and rings assembled.

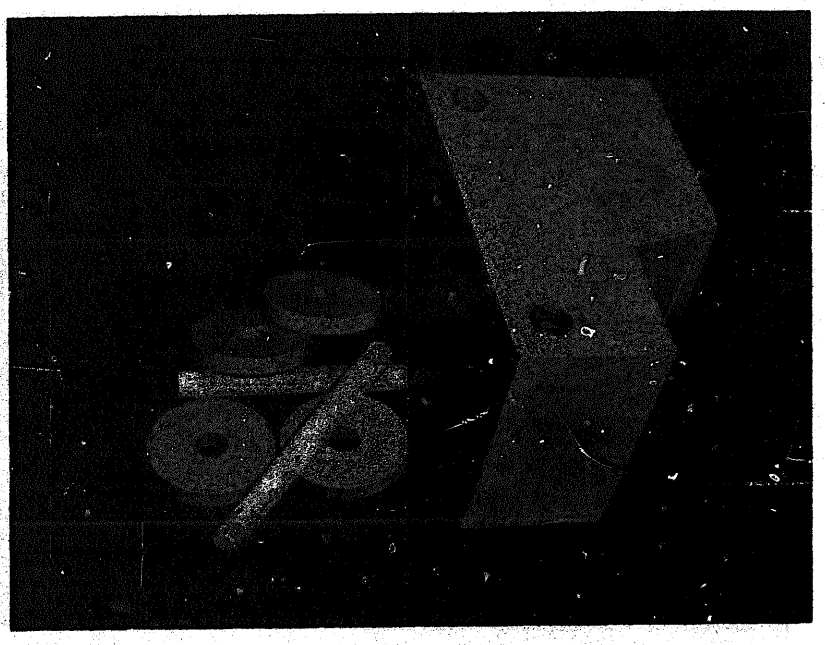

FIG. 4. Parts for the toy car.

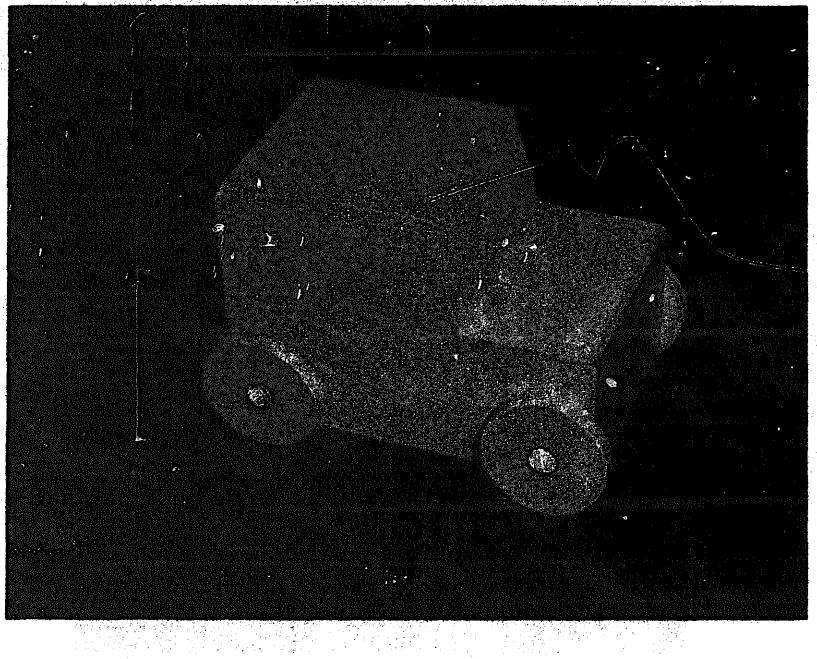

FIG. 5. Toy car assembled.

Our goal has been to develop a system which enables one to:

(i) think up a new assembly involving a kit of parts which have not been used before,

(ii) spend a few hours familiarizing the machine with the appearance and manipulation of the parts and instructing it how to assemble them into the required structure,

(iii) leave the machine unattended, busily making the structures *ad nauseam*, provided that a fresh heap of parts is dumped on the table from time to time.

2.2. Performance

We have achieved our goal for simple structures. We can show the machine half a dozen new parts and instruct it how to lay them out ready for assembly in about two hours; interactively programming the assembly operations themselves takes four hours or so. The machine can make the structures like the peg and rings, the toy car or a toy ship unassisted, but slowly, taking an hour or two to find and assemble the parts. The system occupies about 50K 24-bit words of POP-2 code. It completes the assembly about nine times out of ten.

The machine can deal with mixtures of kits: it has been given the components of the car (7 parts) and of the ship (6 parts) in one heap, and has produced the two complete assemblies.

The program is written as two distinct sub-programs, *layout* and *assembly*, which do not at present communicate with each other, but rely on common conventions.

The layout subprogram depends heavily upon tactile and visual information, and uses internal descriptions of parts and the table top. It can cope with mistakes, failures and accidents.

The assembly subprogram is relatively simple, uses tactile information only and has no internal descriptions of parts. It is principally a rigid sequence of assembly instructions relying on the correct positions of parts on the table or workbench. It cannot recover from accidents, though it can cope with small positioning errors. More interactive and integrated behaviour during assembly is desirable and should be possible, but only with considerable further work.

3. Instructing the System to Perform the Task

3.1. Instructing the layout program

For each part in each stable state (e.g., right way up, or on its side):

(1) The operator places the part on the table under the vertical camera, the machine takes a picture and creates from it an internal description of that view of the part. This is repeated several times, and the machine adjusts

its descriptions each time, taking note of the variations caused by re-orientation of the part and imperfect picture information.

(2) The programmer types in on-line commands to move the hand, to pick up the object, turn it over, put it down and pick it up again if necessary, then to put it down in the standard position and orientation. (If the assembly has several identical parts, separate commands are given for putting each one down in its own place.) The programmer intersperses these commands with instructions to remember the current state, e.g., when the hand has closed over the part. The system takes a note of these specified states and at execute time constructs a sequence of actions to put the part into its standard position and orientation.

3.2. Writing the assembly program

The programmer begins with all the parts laid out in their standard positions and orientations. He then interactively devises and edits some POP-2 program to make the machine pick up the part and fit it into the assembly. His program uses basic move and grasp operations, and two high level manipulation operations provided for constrained moves and hole fitting.

4. Performing the Task

The parts of the kit are dumped in a heap on the platform and the program started by typing RUNTASK. The system transforms the heap into a completed assembly in the following two stages.

4.1. Layout

In this stage the parts are separated and visually identified and then put into their standard positions and orientations.

Parts, or heaps of parts, are first located on the table using the side camera (wide angle). Each part or heap is then inspected with the overhead camera. Recognizable isolated parts are picked up and laid out in standard position and orientation. Unrecognizable objects are called heaps.

If not all required parts have been laid out, the smallest heap is attacked in an attempt to decompose it; if a suitable protrusion is visible it is grasped and pulled from the heap, otherwise an attempt is made to lift it as a whole and turn it over, so that it falls apart; if all else fails the hand is driven through the heap to spread and separate it, if failure is total, attention is directed to another heap.

If some parts are missing, the program complains. If there are more than required it clears them away.

4.2. Assembly

This stage begins with the assumption that all parts have been found and laid out.

The parts are fitted together by feel without any visual guidance and using no internal descriptions of the parts. The sequence of operations is determined by the programmer and can include tactile tests (e.g., to discriminate between visually indistinguishable parts) as well as movements guided by touch (e.g., feeling when a part is firmly located). A workbench with a simple vice and various working surfaces is used.

5. More Detailed Description of the Layout Subprogram

5.1. Its knowledge, specifications and assumptions

There are three classes of information used by the programs: assumptions about the nature of the world (usually in procedural form), specification of certain facts about the world and the robot system (e.g., camera to platform transformations), and representation of the state of the world and the robot.

The most obvious facts about the structure of the robot, namely the existence of platform, hand and TV camera and their operation, are primitive to the system.

The essential parameters of the system are specified by constants or data-structures, including:

range of movements of the hand and platform,
size of the platform, working space, and workbench,
camera transformations, viewpoints, visible space.

The implicit assumptions about the task and parts are: Parts are rigid, small enough to be picked up, and are light in colour. They are also large enough to be visually recognizable and are (usually) visually distinguishable. Parts and heaps are supported by the movable table, which is dark in colour. Parts can each rest on the table in one of a small number (say <10) stable states (a cylinder on its side we define to have one particular stable state, rather than a continuum of states). To each stable state there corresponds a unique view from the overhead camera (unique to within translation and rotation in the image plane).

The number of parts is such that they can be laid out in the designated area without mutual interference.

It is assumed during the layout phase that parts are not fixed together, i.e., are separable by simple pushing or lifting.

5.2. Representation of the world and robot

The state of the robot at any time is given by a record which contains the positions of hand and platform. The camera currently in use, and its characteristics are given by another record. When the picture-taking routine is called, it returns a picture record which contains a grey-scale array, specification of the part of the image sampled, and the records giving the state of the robot and TV camera used.

The rest of the world is specified by a list of all known objects. An object record contains information about the volume of space within which it lies, the type, if known, and its stable state, and various properties, such as "IMMOVABLE" or "PUT AWAY".

Objects records for the hands are created when required and are not kept in the list of known objects.

Descriptions of object types, their stable states and appearances are constructed by the machine during instruction. They are described in detail later.

5.3. Control

The data the program has about the world are held in global variables and data structures. There is a simple top-level routine that examines the data and decides which mode of activity is now appropriate, calling the relevant subroutine. Success or failure of the subroutine causes a return to the top-level test-and-execute loop which is only satisfied when the task is completed. It is therefore persistent in its actions, and always tackles the globally appropriate job.

We may transcribe the POP-2 code thus:

```
loop: if task-completed then goto exit
      elseif all-tidy then assemble
      elseif search-needed then explore (work-area)
      elseif parts-untidy then put-all-away
      elseif extra-objects then discard-objects
      elseif parts-needed then smash-a-heap
      elseif lost-parts then search-for-missing-parts
      elseif extra-heaps then put-heaps-in-corner;
      goto loop
exit:
```

If one of the subroutines encounters difficulties it can simply jump out to loop. Some of the lower level routines set up their own failure traps for particular conditions e.g. some assign a label to the variable PICKUP-FAIL, and a failure to pick up an object subsequently causes a jump out to the current label.

5.4. Exploration

The program relies heavily upon taking a picture and then trying to place what it sees into correspondence with what it knows. It does this by analysing the picture initially as a set of bright regions (blobs) on a dark background. It also computes which known objects may possibly be visible, and the approximate size and position of their images. A matching algorithm (described later) is then used to find the best correspondence between regions

and expected images, allowing for occlusions. The result of this process is an "explanation" of the picture at the level of objects without considering more detailed information such as shape.

During exploration of a space, the program divides the space up appropriately, takes a picture of each subspace with the oblique camera and "explains" the picture. A check of the explanation indicates any hitherto unobserved objects (or the disappearance of already known ones). Hence the program determines which parts of the images require further analysis, and which part of the world needs further examination.

During manipulative activity, visual checks are made that things are proceeding smoothly; discrepancies between what is expected and what is seen invoke appropriate actions (e.g., that there is room to put down an object).

When the program observes an inexplicable region, it generates a new object record, for which it computes an appropriate volume of occupied space, assuming the object to rest on the platform, without analysing its shape in detail. The program takes a detailed picture of the new object (moving the platform if necessary) to make this as accurate as possible. It moves the object under the overhead camera, takes a coarse picture, and finds the appropriate region after making up an explanation. It can then take a more detailed picture of the object, and update the volume occupied by combining the information from the two viewpoints. It is now in a position to apply the recognition process described in Section 6 to the region representing a detailed view of the object from above.

5.5. Separating parts from a heap

The process of removing parts from a heap is heuristic, that is it needs a little luck. But it has never failed, if the machine is left to worry at a heap long enough.

Having decided that a heap is to be broken, the machine chooses the smallest one. It then takes a picture of it with the overhead camera, and checks that it cannot be recognized as a part. (A part may occasionally not be recognized, and therefore be classified as a heap during exploration. If it is subsequently recognized during the check prior to heap-breaking, the program exits from this mode of activity, and will then attempt to put the part away, or discard it.)

The outline of the heap is examined to determine whether a suitable protrusion exists. If there is one which is of appropriate size, a visual check is made to see whether there is room to lower the hands and grasp it; several possible positions are considered. If a graspable protrusion is found, the hand grips it and lifts it away from the rest of the heap. The area near the location of the original heap is now explored to see what has happened.

The program now finds a suitable empty space, by considering its world model and then visually checking the space chosen. The object held is then put down, examined and recognized, if possible. The heap-breaking subroutine is then left.

Should the protrusion grasping go astray, either by fumbling during lifting, or by the protrusion being ungraspable in the first place, another protrusion is considered. Should there be no suitable protrusion left, a second heap-breaking tactic is employed. This second tactic is to feel the height of the heap, and then to make a general grab at it half way up. The hand is lifted, and a tactile check made to confirm that something is being held. If not, a second grab is made, at the bottom of the heap. Having tried the second tactic, if the hand is holding something, the remainder of the heap is examined, and also the object held, as before.

Should the first two tactics fail, the third tactic is to plough the hand through the heap just above the table top, after visually finding suitable empty spaces for the start and finish position of the hand. The area near the heap is now explored. This is effective, but usually disturbs the table top extensively so that considerable updating of internal representation is necessary.

Finally, if the heap has not been broken in two, the machine directs its attention to another heap; it will therefore eventually stop worrying at an unrecognizable object in the belief that it is a heap.

Clearly there are subtler ways of picking objects out of a heap, but our simple tactics suffice.

As a temporary expedient, recognized objects which for some reason have not been successfully grasped during the putting-away mode, are labelled "UNLIFTABLE". The heap-breaking routines can then be applied to them in the hope of extricating them from whatever caused the original failure.

6. Recognition of Parts by the Layout Subprogram

6.1. Descriptions

We shall now explain the kind of internal descriptions of parts used in the layout program. We can then show how it creates these descriptions at instruction time and how it uses them to recognize parts at execution time. For clarity we have simplified a few programming details, glossing over some unnecessary or uninteresting distinctions.

The program works in terms of hierarchical descriptions of the objects on the table. The hierarchy consists of a tree of *entities*; each entity has below it either an n-tuple of entities or a substructure whose nodes are a set of entities with certain inter-relationships. We call these entities at the lower level the *components* of the entity above. In the computer the entities are represented by data records linked by pointers. The entities used are summarized in Table 1, and Table 2 gives a brief description of each.

Each entity either has an n-tuple of components, or it has a set of components. The size of the n-tuple is fixed as in the above "syntax", for example a region has a pair of components (an outline and a hole-set); but the size of the set is not fixed until instruct time, for example the system discovers that the hole-set of a car body side view has two holes.

TABLE 1. Hierarchical structure of entities

An *entity* is either a table top, or an object-set, or an object, or a part, or a hand, or a workbench, or a heap, or a stable state, or a view, or a region, or a hole-set, or a hole, or an outline, or a segment.
A *table top* has an object-set.
An *object-set* has objects.
An *object* is either a part, or a hand, or a workbench, or a heap.
A *part* has stable states.
A *stable state* has a view.
A *view* has a region.
A *region* has an outline and a hole-set.
A *hole-set* has holes.
A *hole* has an outline.
An *outline* has segments.

TABLE 2. The entities used

The *Table top* is the whole collection of things on the table.
An *Object* is any physical thing on the table which can be seen or touched; it is initially distinguished from its surroundings by clear space on the table.
A *Part* is one of the separate pieces needed for the assembly e.g. one of the wheels of the car.
A *Stable-state* is one of the states in which a part can rest on the table, irrespective of orientation or position e.g. on its side, upside down. There should be only a small number of distinguishable stable states.
A *View* is an analysed TV picture.
A *Region* is a connected light area in a picture, possibly with darker holes (we use light objects on a dark background).
A *Hole* is a dark area inside a region.
An *Outline* is the outer boundary of a region or hole.
A *Segment* is a segment of a circle (up to 360°) with specific length and curvature (zero curvature means a straight segment). The irregular boundary of a region or hole is analysed into a small number of segments by curve fitting (Fig. 6).

An entity may possess *properties* and some *relations* (in practice only binary ones) may subsist between its components. The properties and relations have names and may be truth-valued or take values in some other domain such as integers or else reals with a given standard deviation. For example, properties of regions are area and compactness (area/perimeter^2), and relations between segments are distance and relative orientation.

We make an important distinction between three kinds of entities, *sense data*, *model* and *individual*. Each has entities of its own kind as components.

A *sense data* entity is an internal description of information obtained from the TV camera, generated by a particular exposure to a physical object. A *model* entity is a summary or composite of a number of such experiences. The end result of the instruction phase is a collection of model entities incorporating the system's knowledge about the parts, their views, outlines, etc. and their variations. The sense data views and outlines which gave rise to them will have been discarded.

FIG 6. Segmentation of an outline.

The important operation in recognition is the finding of a match between sensory and model entities, determining which components of one correspond to which components of the other. The match is expressed as an *individual* entity which contains a pointer to its sense data and one to its model; thus binding them together. The components of an individual are individuals which in turn bind corresponding components of sensory and model entities. An individual also contains certain specific information, e.g. position and orientation, not appropriate to model entities.

There are a few exceptions to the above. The system does not create models for its hand or the workbench at instruction time; these are given beforehand. There is no model for a heap since an individual heap is generated by elimination, on failure to recognize a part, hand or workbench.

The system has a data structure for each sense data entity, model entity and individual entity; these are POP-2 records linked into a tree structure by pointers to their components. There is also a data structure for each *entity class*, for example the class "view" and the class "region". Each

sensory, model or individual entity belongs to some class and certain data pertains to the class as a whole—namely (i) a list of the properties which an entity of that class enjoys and functions for computing their values from the corresponding sense data, (ii) similarly for relations, (iii) functions to extract the sense data for components from the sense data for the whole (e.g., to extract the starting and finishing points, curvature, etc. for each component segment from the chain encoded representation of the boundary).

The entity classes are one level of generalization above model entities. No matter what kit the program is working with, it has the same basic hierarchy of parts, views, outlines, etc. The data structures representing the entity classes provide the program with the knowledge needed to construct the hierarchy of model entities.

Table 3 summarizes the four notions of entity class, model entity, individual entity and sense data showing what information is associated with each. Fig. 7 gives the data structures representing corresponding entities in a simple case. The use of this information will be clearer when we discuss the recognition process.

TABLE 3

Notion and symbol	Associated descriptive information
Entity classes, C	Property names P, Relation names R, Property finding functions $f: D \to V$ for $p \in P$ (V is the set of values for properties and relations). Relation finding functions $f: D^2 \to V$ for $r \in R$, Classes of components, Component finding functions in $D \to D$, matching function in $D \times M \to 2^I$.
Model entities, M	Class, values of properties and relations, components.
Individual entities, I	Class, values of properties and relations, components, parameters, points to model and to sense data.
Sense data, D	At view level: 2-D brightness array from TV picture. At region level: region perimeter as list of vector increments, 2-D Boolean array showing whether point is inside and brightness array. At hole level: as for region level referring to dark holes. At segment level: segment length, curvature and position.

(Note: $f: X \to Y$ means f is a function from X to Y, and $\times$ means Cartesian product.)

FIG. 7. Hierarchies.

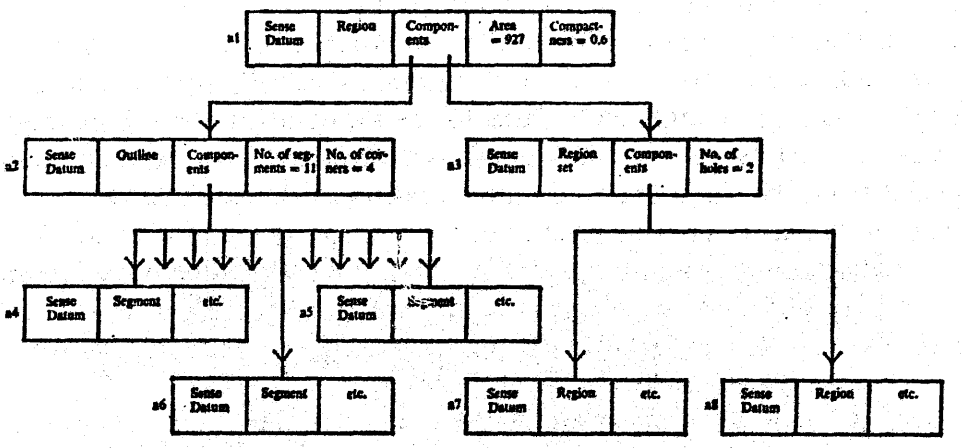

FIG. 7a. A hierarchy resulting from analysis of T.V. data.

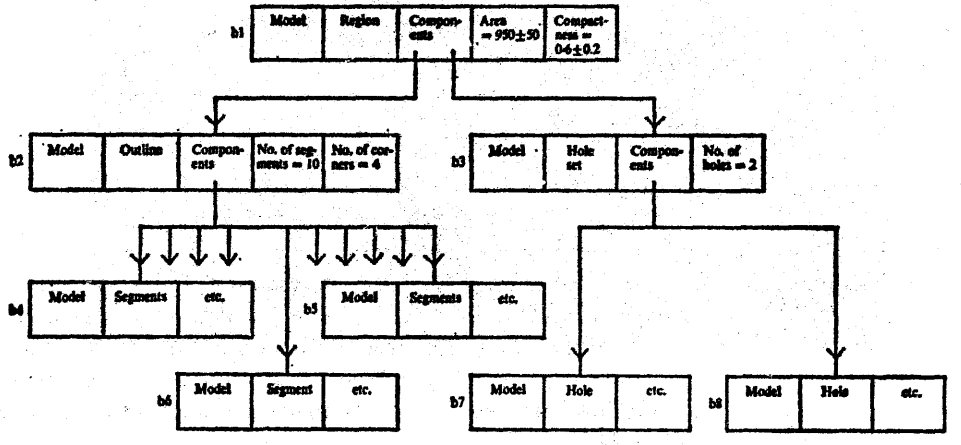

FIG. 7b. A hierarchy for a model region.

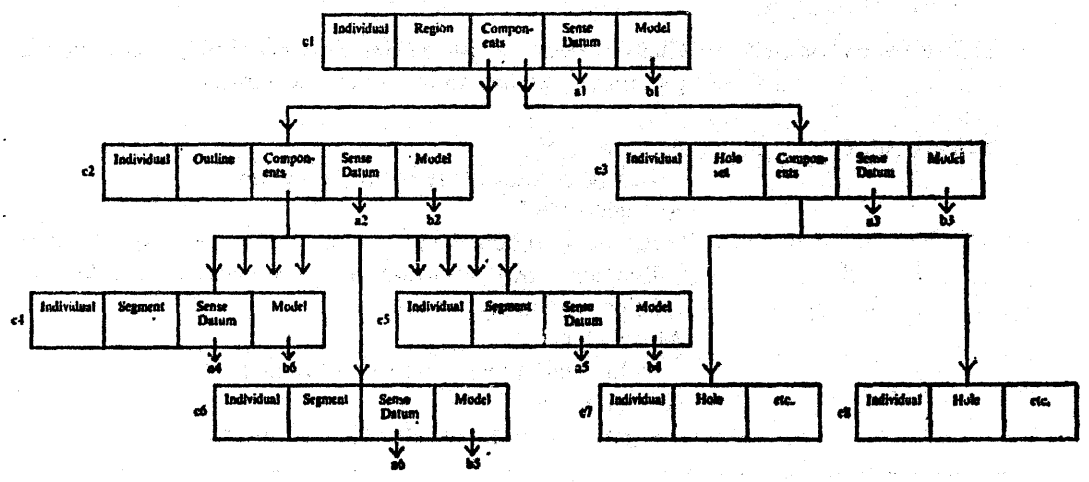

FIG. 7c. A hierarchy for an individual region, binding sense data to models.

6.2. The matching process which recognizes parts

To understand the recognition process let us consider what happens when the system has taken a TV picture and tries to interpret it as a side view of a car body. This may be during the instruction phase when it has been told that it is looking at the side view of a car body, or during the execution phase after it had found an upright car body and turned it over. Or again it might be dealing with an unknown object, and "sideview of car body" might be just one possible interpretation among several which it was trying.

The TV picture, a 2-dimensional array of brightness levels, constitutes a sense datum, d in D. A matching function is now applied to this sense datum and the model of the side view of the car body. This function produces a set of individual side views of car bodies, an empty set if there is no way of interpreting the picture as such a view, otherwise one element for each possible interpretation. Thus

match: Sense data × Models → Set of individuals.

The matching function works its way recursively down the hierarchy from top to bottom, comparing gross properties on the way down and failing if they are too discrepant. Thus it might fail because the area of the region it is looking at in the picture is too small for a car body, without bothering to analyse the outline of the region. As it goes down it refines the sense data, using a thresholding region finder routine to get region level sense data from the view level brightness array, finding holes with the same routine to get hole data and fitting curves to the perimeters to get segment data. (We have called the region finder and curve fitter "component finding functions".) When it gets to the bottom level the recursion unwinds and passes up the hierarchy descriptions of individual entities, using their finer properties and relations between them to establish correspondence with the model and to resolve some ambiguities. At each level the matching function produces a *set* of individual entities each of which might correspond to the model, thus dealing with ambiguity essentially as would a "back-track" or non-deterministic process.

To be more precise, the function "match" works as follows:

Function match(d, m)
  Let c be the entity class of m.
  Let f be the special matching function of class c.
  Result $= f(d, m)$.

The special matching function f may vary from class to class, but normally f is "general-match", defined as follows:

General-match: Sense data × Models → Sets of individuals
Function general-match(d, m)
  Let c be the entity class of m.

Let P be the set of properties for class c.
For each p in P, compare $f_p(d)$, the value of property p for the sense datum, with the value of p in the model m. If there is too much discrepancy exit with *result* = empty set.

Case 1. m has an n-tuple of components, $m_1, \ldots, m_k$. Apply the component finding functions of the class c to d, to find sense data relevant to the components say $d_1, \ldots, d_n$. As each d_j is computed, match it against the component m_j, thus let $I_j = \text{match}(d_j, m_j)$. If some I_j is empty, then exit immediately with *result* = empty set. Otherwise use each element of $\Pi_{j=1}^{n} I_j$ (the Cartesian product of the set of individual components) to construct a new individual with these components. The *result* is the set of these individuals.

Case 2. m has a set of components, S_m. Apply the component finding functions for class c to find a set of sense data relevant to the components, S_d. Use the relation value finding functions of c, f_r for r in R, to compute the values of relations between the S_d. Compare these with the known values of the relations for the S_m, and use the relational structure matching algorithm (described below), together with the function *match*, to find the largest subsets of S_d which correspond. If these subsets are sufficiently large, construct a new individual entity from each correspondence among the components. The result is the set of individuals so constructed.

Our recognition process is basically recursive (we have experimented with some process-swapping techniques but they are not in our program at present). This does not lead to too much rigidity, because we make rather extensive use of "memo-functions" (Michie [11]); for example, when an analysis of the outline of a region is required the analysing function remembers the result and, when asked for it again, simply returns it immediately. Thus outline analysis is only done when needed and never repeated.

6.3. Learning and modifying descriptions

The description of an object is "learnt" by the system being shown that object underneath the vertical camera, and being told its name and current stable state. Using the taxonomy to guide it, a model entity is made, the regions of the view, the holes, and the outlines of the regions and holes being computed, together with their properties and the relevant relations.

The learnt description is modified by showing the system other views of the same object in the same stable state. In order to modify a model, the new view is matched with the model, exactly as in the recognition process (but with some constraints loosened). If a successful match is made, those parts of the model which have been matched in the individual are updated.

The process is repeated until all possible "stable states" of the object have been observed.

6.4. Matching relational structures

A set of segments forming the outline of a part can be regarded as a relational structure endowed with properties, such as length and curvature, and relations, such as adjacency, distance or relative orientation. Holes and objects on the table top can be similarly regarded. Although the properties and relations usually take numerical values (length) it will simplify the discussion to talk in terms of truth valued ones (long, medium, short). In the

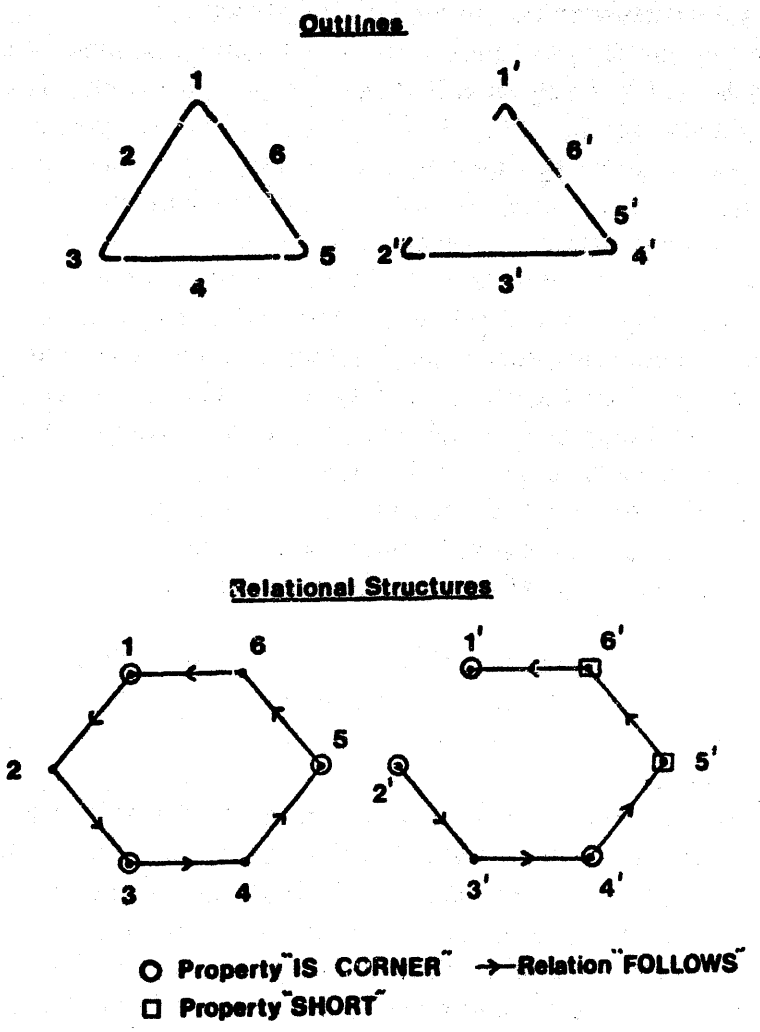

FIG. 8. Outlines and their corresponding relational structures.

algorithm given in Section 6.2 there is a point (Case 2) where we need to put two sets (of segments, say) into correspondence on the basis of these properties and relations. TV picture processing being what it is we expect discrepancies (segments missing, two segments coalesced) but we want to match as many elements as possible. Fig. 8 shows two simple outlines and corresponding relational structures; they have several common substructures, e.g., {11', 32', 54', 43'}.

More precisely, by a *relational structure* we mean a set of elements together with a set of properties P and a set of relations R over it (we consider only binary relations here). Given two relational structures $\langle S_1, P, R\rangle$ and $\langle S_2, P, R\rangle$ we define a match between them as a set $T_1 \subseteq S_1$, a set $T_2 \subseteq S_2$ and an isomorphism, $\approx$, between T_1 and T_2 preserving properties and relations. Thus $s_1 \approx s_2$ implies $p(s_1)$ iff $p(s_2)$ for each p in P, also $s_1 \approx s_2$ and $s_1' \approx s_2'$ imply $r(s_1, s_1')$ iff $r(s_2, s_2')$ for each r in R, and a match represents a common substructure in our two relational structures.

We can find matches as follows. By an *assignment* we mean a pair $\langle s_1, s_2\rangle$ with s_1 in S_1 and s_2 in S_2 such that $p(s_1)$ iff $p(s_2)$ for each p in P. In Fig. 8 the assignments are: 11', 12', 14', 23', 31', 32', 34', etc. We say two assignments $\langle s_1, s_2\rangle$ and $\langle s_1', s_2'\rangle$ are *compatible* if $r(s_1, s_1')$ iff $r(s_2, s_2')$ for all r in R. Now by definition a match is just a set of assignments such that each assignment is compatible with every other assignment in the set. Indeed we may think of the assignments as forming the nodes of a graph with compatibility as the (symmetric) relation forming the arcs. Our problem then is to find totally connected subsets of this graph, often called *cliques*.

A clique is said to be maximal if no other clique properly includes it. Finding maximal cliques is a well known problem (Karp [10]). A graph of n assignments may have $2^{n/2}$ or $3^{n/3}$ maximal cliques in a theoretical bad case, but at least we can find each maximal clique quickly. We can do this by using a refinement of a simple binary search algorithm given by Burstall [5]. Bron and Kerbosch [4] present an algorithm which differs only slightly from ours.

In fact, for our recognition problem we only wish to find largest maximal cliques (those with the largest number of elements). We can readily guide our algorithm to do this by introducing a maximizing search, instead of simple recursion.

In this section we have defined an assignment as a pair of elements, one from each structure, which have identical property values. More loosely one can demand only some degree of similarity in the properties. Our program uses as the assignments the set of all pairs $\langle m', i'\rangle$ such that i' is in $match(d', m')$ for some m' in S_m and some d' in S_d. This gives us the nodes in our graph; the arcs are found by defining compatibility to be some suitable degree of similarity in the relations, not necessarily demanding one-one correspondence.

This relational structure matching is used three times by the layout program: to match segments in an outline, to match holes in a region and to match objects on a table top, e.g., associate some objects which it now sees with its previous knowledge. It gives us a robust matching technique which can make correspondence between observed and predicted elements in spite of imperfect data. It has some similarity in its way of working to

Waltz's technique for labelling pictures (Waltz [15]). Winston [16] also does structure matching. Graph isomorphism is a well-known problem (e.g., Corneil and Gottlieb [7]); relational structure isomorphism is essentially the same. Testing whether one structure is a substructure of another (graph injection) is computationally harder (Barrow et al. [2]), indeed it is known to be polynomial complete (Cook [6]); our problem here, finding common substructures, is still more onerous, but it is needed if both structures have imperfections.

6.5. Motivation and development of the above approach to matching

The entity taxonomy evolved during our programming. At first we tried evading most of the difficulties of visual recognition by using white objects and black background, and viewing from overhead; we hoped a simple technique like template matching would cope well enough for us to direct our attention to other problems, at least temporarily. In the event, experiments with matching boundaries of regions (slightly smarter than templates) were frustrated by camera non-linearities and perspective distortions, so we decided to use structured descriptions instead of pictorial data.

Some of our previous research (Barrow et al. [2]) had been concerned with the problems of matching such descriptions—simple techniques explode combinatorially, so some subtlety is called for. A powerful approach is to decompose the description into pieces that can be treated more or less independently, a small number of matches found for each, and then some of these are combined to find a match for the whole. This approach works well when descriptions have an obvious decomposition (e.g., a teapot must have a handle, a body and a spout), particularly if the decomposition can be made recursively and if substructures (like spouts) are common to a number of descriptions—in which case redundant work need not be done.

In our next experiments, we formed such hierarchical descriptions of views of objects and wrote programs in which the knowledge was embedded procedurally; that is, the description and matching were mixed together as a sequence of tests to be made. The subroutine structure of the program reflected the decomposition of the description.

While we were able to make this scheme work, we found a number of problems: programming was tedious, the representation was rather inflexible, suitable decompositions were not always unique or obvious. The problem of imperfect matches had not been adequately solved (e.g., if the tip of the spout is missing, that could mean no spout and hence no teapot, despite much evidence to the contrary). We wished to alleviate these difficulties without incurring combinatoric penalties, and so were led to the present scheme.

Firstly, we noted that there was an indisputable hierarchical decomposition

of our view descriptions, namely that of data types, regions, outlines, holes, etc. which it would be foolish to ignore. Secondly, we should use properties as a primary test; relations hold between entities at the same level (i.e., between the components of the entity of the level above). Thirdly, our hierarchy should operate primarily "top-down" since if an entity cannot be matched by virtue of its properties, its components clearly do not match.

We were faced with one remaining problem: at some levels we must match a set of things against another set, for example a set of holes, a potentially explosive situation. Any decomposition into subsets would be arbitrary, and therefore prone to problems outlined above. We therefore reconsidered the matching problem and transformed it, as we demonstrated earlier, into that of finding cliques of a graph. Since no suitably efficient clique-finding algorithms were available we devised our own.

Matching via cliques seems a good technique; it is tolerably efficient and can give partial matches for misinterpreted or imperfect information (e.g., occlusion), in a pseudo-parallel way. Since set matching still tends to be exponential in complexity, the hierarchy is necessary to keep sets small.

We rewrote there cognition system in line with these ideas, in particular separating description data from matching procedures. In order to simplify programming we arranged for the system to be capable of learning descriptions. This we achieved by providing a data structure describing the entities (regions, holes, etc.) and giving information about how to find, represent and match them. Thus provided with a "model of models" it is relatively straightforward to construct particular models of views automatically. We have the additional bonus that we can readily change the nature of the hierarchy and its component entities by changing the defining data structure, but we can still use the same program to construct and match models.

Having produced an operational recognition system, it became clear that its underlying scheme could be extended upwards to include objects and the table top. The matching techniques were used to "explain" each picture taken in terms of objects, but the hierarchy was never extended to include this process uniformly. Above the level of views *ad hoc* code was used.

We think any scheme which simply finds possible correspondences between descriptions, without accounting for the discrepancies in some way, unsatisfactory. Unaccountable discrepancies indicate an incorrect correspondence. The process that explains pictures of the table top does try to account for unexpected phenomena such as occlusions, missing objects, new objects, and does suggest appropriate action to be taken—e.g., go and examine that new object. The recognition process does not (beyond accounting for anything not matched as a "heap"). We considered trying to improve a correspondence between a pair of segment sets by breaking or merging

segments, but decided (a) that the models we were using were not good enough (see Section 8.3) to enable anything useful to be done, and (b) that the time required to do this for a doubtful correspondence would be better spent taking and examining another picture of it.

7. More Detailed Description of the Assembly Subprogram

This program works blind, using only hand sensing. It is written interactively at instruction time in terms of basic hand moving and sensing operations together with two higher level operations.

The basic operations include "raise z centimetres", "move to (x, y)", "grasp to w centimetres", "rotate palm by θ", and functions for reading forces on the hand by means of strain guages, namely gripping force, weight of the object held and torque. There are two higher level operations: *constrained move* and *hole fitting*.

Constrained move. This operation has two parameters, both force vectors, f_m a force opposing movement and f_c a constraining force (Fig. 9). Let u_m and u_c be the unit vectors in these two directions. Let ε_m and ε_c be small scalar distances. The hand attempts to move in direction $-u_m$ until it is opposed by a force larger than f_m. At the same time it keeps in contact with a surface which offers a resisting force f_c. The resulting movement will not necessarily be in direction $-u_m$ but along the surface according to the component of $-u_m$ tangential to the surface. The operation works in detail as follows:

(1) If a force greater than f_m is felt, then stop.
(2) Move by $-\varepsilon_m u_m$.
(3) If a force greater than f_c is felt then move by $\varepsilon_c u_c$ and *goto* 3, otherwise *goto* 1.

Fig. 9 shows an example of the pattern of movement which a single call on the constrained move operation might produce. The constraining force parameter may be left undefined giving an unconstrained move in direction $-u_m$ until an opposing force is felt. This operation was suggested to us by work at MIT Draper Laboratory (Nevins et al. [13]), which gave us several valuable ideas about assembly.

Hole fitting. This operation has two force vector parameters f_c and f_s. Let u_c and u_s be the corresponding unit vectors and ε_c and ε_s small scalar distances. It is used when the hand is holding an object with a protrusion which must be fitted into a hole (e.g., axle into car body) or is holding an object with a hole which is to be fitted over some protrusion (e.g., putting a ring on the peg). The hand moves in a spiral pattern normal to u_c, pushing repeatedly in the direction u_c until an opposing force greater than f_c is felt

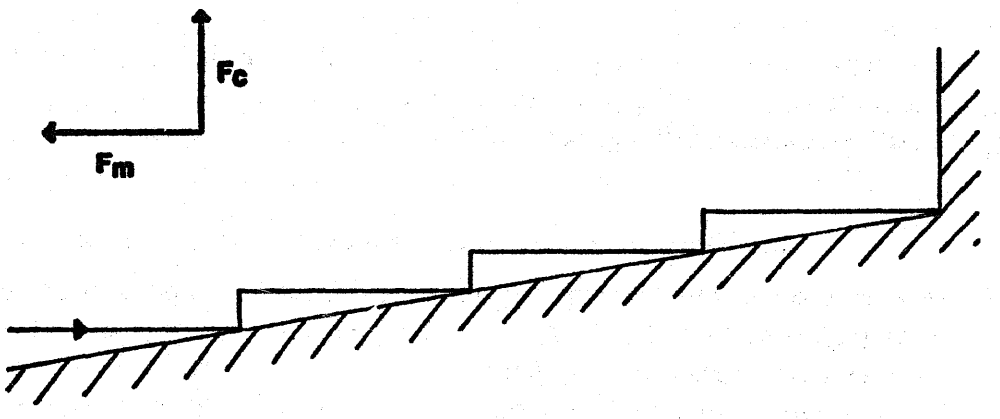

FIG. 9. A constrained move.

FIG. 10. Assembling the car on the workbench.

and then retreating and making a short move in the direction of the spiral before trying again. When, at some point in the spiral, it succeeds in moving forward by $\varepsilon_c \mu_c$, it tests whether it has found the hole by moving sideways by $\varepsilon - {}_s\mu_s$. If a force greater than f_s is encountered (or after five attempts), the process stops, otherwise the spiral pattern is resumed.

The tactile hole fitting program allows the robot to make insertions which could not be made by dead reckoning either because of defects in the robot's absolute position sense or because of irregularities in the bodies being handled. The effective gain is not large—it is feasible to search an area 4 mm square with a pitch of 1 mm. The technique is dependent on being able to maintain sufficient local dead reckoning (in space and time) to accomplish the pattern.

The wooden car assembly gives an idea of how the program proceeds. A "workbench" is used, fixed to one corner of the table (Fig. 10). It has a "vice" for holding a wheel while an axle is being inserted, consisting of an L-shaped corner piece and a pivoted bar which the hand closes so that the wheel is held between the bar and the L. It also has a vertical "wall" so that the car body can be held firmly while the second wheel is pushed onto each axle.

The sequence of events, in outline is:

(i) The hand puts a wheel in the vice and inserts an axle.

(ii) It turns the car body upside down, picks up the axle with the wheel on it and inserts it into the body.

(iii) Repeat (i) and (ii) for the second wheel and axle.

(iv) Put the car body against the wall upside down with the two wheels against the wall.

(v) Push the remaining two wheels onto the protruding axles.

(vi) Pick up the assembled car and place it on the table.

Assembly programming is still quite tedious, involving choice of numerical parameters for distances and forces, and we have some ideas for easing it (Ambler and Popplestone [1]). There is clearly a lot of thinking to be done before we could make the assembly phase as versatile and easily instructable as the layout, e.g., by replacing numerical commands with instructions using relations like "on top of" and "fitting into" or by showing the machine intermediate assemblies. In particular our present assembly subprogram does not use the internal descriptions of the parts which have been acquired during instruction by the layout program. Such descriptions would have to be recast so as to be useful for assembly as well as recognition.

8. Concluding Remarks

8.1. Development

The project took one year (about four man-years effort) to complete with approximately the following breakdown

	man-year
basic facilities (extension)	1
recognition	1½
layout	⅓
heap-breaking	⅓
assembly	⅓
the rest (e.g., teaching, testing)	½

Each part of the system was programmed by one of us, and the parts were put together later.

At the beginning, it became clear that the existing software packages for vision and manipulation needed to be rewritten, the vision package to deal properly with multiple cameras, and mobile viewpoint, the manipulation package to deal with force feedback. It took some time to hit upon a good representation for concepts like cameras, pictures and robot positions and their interrelationships.

Much effort (perhaps 30%) went into the development and implementation of the recognition processes. As indicated in Section 6.5, several versions were tried before the current one was developed. Initially the recognition system was an independent package; the function RECOGNIZE was given a region, and returned an individual. Its development was therefore relatively unconstrained. It was only much later that some of the recognition routines were used to explain every picture taken. In the current system, the recognition package still remains self-contained. However, we have recently experimented with manipulation as an aid to vision; height of an object is easily measured by lowering the hands onto it; an attempt can be made to turn over an object when an ambiguous view is seen so that another picture can be taken. To some extent these actions can be integrated into the hierarchy but further thought is necessary to do so cleanly.

The heap-breaking routines were also developed as an independent package; they do not call the recognizer, they originally used only basic facilities. It was only when the pieces of the system were put together that they were integrated a little more, checking that the heap is unrecognizable before attacking it and using the "world model" to choose a place to put down an object grabbed.

The assembly procedures remain an independent package. They use no vision, and do not even use the world model for specifying location of parts initially. In fact, only the basic manipulative procedures are shared with the rest of the system. The assembly routines represent only about 10% of the complete system.

Putting the pieces of the system together was remarkably trouble-free. We can attribute this mainly to the gross modularity of the system and the in-

dependence of the component pieces, and partly to daily interaction of those of us writing the program.

A master package which called the recognizer, heap-breaker and assembler (and included the rest of the system—exploration, laying out and clearing away routines) was written. It seems to contain most of the more messy processes, such as maintaining the world model and dealing with disasters, and most of the interactions. Initially the master package was written with very simple subroutines representing the other packages, which were eventually slotted in later. Afterwards, however, some attempt was made to integrate the packages by eliminating duplicated procedures, generalizing procedures, sharing representations and so on, but there is still scope for further rationalization and integration.

8.2. Performance

Despite its apparent crudeness, the system works quite well. It usually manages to accomplish its task successfully. It is apparent that the layout phase is more robust than the assembly phase. We can attribute this to three factors. Firstly the layout routines use much feedback during their activities, particularly visual input (the system typically takes up to 150 pictures in assembling the car and ship simultaneously). The assembly routines, however, fail if contact with a part is lost. Secondly, the layout routines maintain a flexible world model and a more explicit representation of objects, etc. The assembly routines have no models which can be used for planning, prediction, etc., nor updated; all information is embedded in the program in an opaque manner. Thirdly, while the assembly routines can recover from minor positional errors, recovery from gross errors is very difficult. The layout routines do not have the intelligence or ability to break up a half completed assembly, nor even to recognize it as such, even less so the assembly routines.

The robustness and dexterity of the layout routines lie mainly in the control scheme: it is *very* persistent, and it does not abdicate too much responsibility to subordinates. The decision what to do next is made frequently, and on the basis of all the available information, so the system never persists with an inappropriate task but uses serendipity.

It seems that to try to use backtracking, co-routining or multiple processes would actually be less appropriate and less effective, and probably not even easier. While we were writing the system, we occasionally thought that a higher level language, such as PLANNER or CONNIVER would have been advantageous. Looking at the completed system, it seems that the facilities such languages provide would not be much exercised. The representation of the world is particularly simple, and the control structure is too. POP-2 was a very good language for writing this program; its facilities were precisely

what were needed. On the other hand, the structure of the system might have been very different if a higher level language had been used.

8.3. Improvements

Having completed the project, in what ways could we improve upon the system?

Certainly the most significant simplification we made was the avoidance of the 3-dimensional aspects of the problem, except in the very simple case of computing the "volume" (i.e., enclosing box) occupied by an object. To deal with these adequately it is necessary to consider:

(1) How to model 3-dimensional objects, and assemblies.

(2) The relation between the 3-D shape of an object or an assembly, and its appearance from an arbitrary viewpoint.

(3) The synthesis of a 3-D description of an object from several different views of it.

(4) Making the system "understand" the effects of various handling operations on the actual position of an object, and therefore able to reason about the result of grasping, tilting, moving, etc.

(5) The description of the assembly process in general terms as a sequence of statements about relations to be made to hold between features of objects, rather than as a sequence of program instructions involving numerical coordinates.

Work is currently in progress on these problems, but it will be appreciated that each requires a major project of its own.

The system at present does no planning, apart from choosing an empty space into which to put down an object. In fact there was little need for planning, in the sense of reasoning and making hypotheses; most decisions have an obvious or easily computable best alternative. One may argue that there has been too much emphasis in the field of A.I. upon planning and "problem-solving", and not enough on "doing". The current wave of A.I. languages seems to reflect this, as we remarked earlier. Perhaps "doing" is just harder to work on, with more effort necessary before demonstrable success, and is consequently more expensive. It is certainly a strain to remain true to one's scientific ideals while being criticized simultaneously for not using tricks and short-cuts to make things faster and cheaper, and for using tricks and short-cuts to constrain the problem domain!

Producing the system raised one or two issues, other than 3-dimensionality, which are worth further investigation. One is the clique-finding approach to recognition, which arose spontaneously from the project. The approach has many virtues, pseudo-parallelism, non-localness, and is actually closely

related to other domains, such as constraint satisfaction, inference making and programming languages.

The process for automatic learning of descriptions is unsatisfactory, since undue importance is given to the first view of an object. A subsequent view *does* modify a model, but only those parts of it are used which actually match the existing model. We would like to extend the idea of matching pairs of relational structures to matching sets of such structures in order to make better models. No work has been done on this.

The representation of the world (objects, assemblies, robots, tools, etc.) and the use and maintenance of the representation is something that our present project barely touched upon. It is worthy of a great deal more work.

Lastly, the control structure of a system which has a high degree of interaction with the outside world (expectations, surprises, and very high information transfer rates) has been hitherto rather ignored. It too deserves further research.

Writing the program has provided a useful insight into the problems important to an integrated assembly system. By tackling a definite, but deliberately simplified, task we hav explored the ways in which full advantage might be taken of the apparatus and programs available to us (e.g., using the hand to make the vision problems simple ones). The requirement of versatility has meant that we have had to introduce methods of learning by example and therefore the construction and generalization of models.

Our primary intention in writing this versatile assembly program was to explore the problem area, making certain simplifications in order to evade currently difficult problems. We regard the present system as a zero-order version with considerable room for improvement, but it is clear that much has been accomplished with relatively simple ingredients put together in an appropriate way.

ACKNOWLEDGMENTS

The robot equipment was built by S. Salter of the Bionics Laboratory and G. Crawford as part of a robotics project under the direction of Professor D. Michie, and we would like to express our appreciation of their efforts. We are grateful to K. Turner for contributing to picture processing software, and to Eleanor Kerse for typing. D. Bobrow made helpful comments about a draft.

This work was supported by the Science Research Council.

REFERENCES

1. Ambler, A. P. and Popplestone, R. J. Inferring the positions of bodies from specified spatial relationships. *Artificial Intelligence* 6 (1975) 157–174.
2. Barrow, H. G., Ambler, A. P. and Burstall, R. M. Some techniques for recognizing structures in pictures. *Frontiers of Pattern Recognition*, S. Watanabe (ed.), Academic Press, New York (1972), 1–29.

3. Barrow, H. G. and Crawford, G. F. The Mark 1.5 Edinburgh robot facility. *Machine Intelligence* 7, B. Meltzer and D. Michie (eds.), Edinburgh Univ. Press, Edinburgh (1972), 465–480.
4. Bron, C. and Kerbosch, J. Algorithm 457. Finding all cliques of an undirected graph (*H*). *Comm. ACM.* **16** (9) (September 1973).
5. Burstall, R. M. Tree-searching methods with an application to a network design problem. *Machine Intelligence* 1, N. L. Collins and D. Michie (eds.), Oliver and Boyd, Edinburgh (1967), 65–85.
6. Cook, S. A. The complexity of theorem-proving procedures. *Proc. Third Annual ACM Symposium on the Theory of Computing*, Shaker Heights, Ohio (1971), 151–158.
7. Corneil, D. G. and Gottlieb, C. C. An efficient algorithm for graph isomorphism. *J. ACM.* **17** (1970), 51–64.
8. Ejiri, M., Uno, T., Yoda, H., Goto, T. and Takayasu, K. An intelligent robot with cognition and decision-making ability. *Proc. Second International Joint Conference on Artificial Intelligence*, London (1971), 350–358.
9. Feldman, J. A. Private Communication (1973).
10. Karp, R. M. Reducibility among combinatorial problems. *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher (eds.), Plenum Press, New York (1972), 85–103.
11. Michie, D. "Memo" functions and machine learning. *Nature* **218** (1968), 19–22.
12. Michie, D., Ambler, A. P., Barrow, H. G., Burstall, R. M., Popplestone, R. J. and Turner, K. Vision and Manipulation as a Programming Problem. *Proc. 1st Conference on Industrial Robot Technology*, Nottingham (1973).
13. Nevins, J. L., Sheridan, T. B., Whitney, D. E. and Woodin, A. E. The multi-moded remote manipulator system. E-2720, Charles Stark Draper Laboratory, MIT, Cambridge, Mass. (1972).
14. Salter, S. H. Arms and the robot. Bionics Report No. 9, Bionics Research Laboratory, School of Artificial Intelligence, University of Edinburgh, Edinburgh (1973).
15. Waltz, D. L. Generating semantic descriptions for drawings of scenes with shadows. Artificial Intelligence Laboratories Report AI TR-271, MIT, Cambridge, Mass. (1972).
16. Winston, P. R. Learning structural descriptions from examples. Artificial Intelligence Technical Report 213, Artificial Intelligence Laboratory, MIT, Cambridge, Mass. (1970).
17. Winston, P. R. Wandering about on the top of the robot. Vision Flash 15, Artificial Intelligence Laboratory, MIT, Cambridge, Mass. (1971).
18. Winston, P. R. The MIT robot. *Machine Intelligence* 7, B. Meltzer and D. Michie (eds.), Edinburgh Univ. Press, Edinburgh (1972), 431–463.

Received July 1974; revised version January 1975