```
@device(postscript)
@make(article)
@Font(TimesRoman)
@Style(LineWidth 4.24inches, Spacing 1)
@use[Bibliography="<henderson.proposals>general.bib"]
@use[Bibliography="<hansen.papers>myown.bib"]
@begin(format)


@MajorHeading(The Specification of Logical Sensors)@+(1)@foot(This work was suppo:
ted in part by NSF Grants ECS-8307483, MCS-82-21750, DRC-850639, and DMC-8502115)



@begin(center)
@b(Tom Henderson)

Department of Computer Science
The University of Utah
Salt Lake City, Utah 84112

@Value(date)



@end(center)









@center[@b(Abstract)]

@end(format)
Multi-sensor systems require a coherent and efficient treatment of the
information provided by the various sensors.  We propose
a framework, the Logical Sensor Specification System, in
which the sensors can be abstractly defined in terms of computational
processes operating on the output from other sensors.  Issues addressed
include: (1) integration of various kinds of sensed
data, (2) sensor system reconfiguration for fault tolerance or dynamic
context, and (3) the control of sensors.  Various properties
of such an organization are given, and a particular implementation
is described.
@newpage()
@Section(Introduction)
In order to achieve more reliable and autonomous systems (e.g.,
autonomous vehicles and remote space stations) complex sensor systems and
controllers are being developed.  However, the dynamic nature of such
systems, and the random nature of much of the data make it very difficult
to foresee or simulate the behavior of such systems.  There is also the
problem that the system which is eventually produced may not in fact be the
required system.  Our goal is to develop a methodology which permits the
design and specification of sensor systems such that their properties can
be analyzed (semi-)automatically.  Moreover, we believe that it may be
possible to synthesize such specifications based on a knowledge of the
system requirements and the available sensors and algorithms.
We base the sensor system specification on the notion of Logical Sensors
which characterize, in addition to the the block box properties of the
sensor, a certain amount of internal structure.
The overall goal of Logical Sensors and the Logical
```

Sensor Specification Language is to aid in the coherent synthesis of efficient and reliable sensor systems (see@cite[Hansen83,Henderson84a,Henderson84c,Henderson85h,Shilcrat84]).

Both the availability and need for sensor systems is growing, as is the complexity in terms of the number and kind of sensors within a system. Unfortunately, most robotic sensor-based systems to date have been designed around a single sensor or a small number of sensors, and @u(ad hoc) techniques have been used to integrate them into the complete system and to operate on their data. In the future, however, such systems must operate in a reconfigurable multi-sensor environment; for example, there may be several cameras (perhaps of different types), active range finding systems, tactile pads, and so on. In addition, a wide variety of sensing devices, including mechanical, electronic, and chemical, are available for use in sensor systems, and a single sensor system may include several kinds of sensing devices. Thus, three issues regarding the configuration of sensor systems arise:

@begin(enumerate)
How to develop a coherent and efficient treatment of the information provided by many sensors, particularly when the sensors are of various kinds.

How to allow for sensor system reconfiguration, both as a means toward greater tolerance for sensing device failure, and to facilitate future incorporation of additional sensing devices.

How to control sensor systems.
@end(enumerate)

The @u(M)ulti-sensor @u(K)ernel @u(S)ystem (MKS) has been proposed as an efficient and uniform mechanism for dealing with data taken from several diverse sensors@cite[Henderson83,Henderson83a,Wu83]. MKS has three major components: low-level data organization, high-level modeling, and logical sensor specification. The first two components of MKS concern the choice of a low-level representation of real-world phenomena and the integration of that representation into a meaningful interpretation of the real world, and have been discussed in detail elsewhere@cite(Wu83). The logical sensor specification component aids the user in the configuration and integration of data such that, regardless of the number and kinds of sensing devices, the data is represented consistently with regard to the low-level organization and high-level modeling techniques that are contained in MKS. As such, the logical sensor specification component is designed in keeping with the overall goal of MKS, which is to provide an efficient and uniform mechanism for dealing with data taken from several diverse sensors, as well as facilitating sensor system reconfiguration. However, the logical sensor specification component of MKS can be used independently of the other two MKS components; for example, in conjunction with any desired low-level organization and high-level modeling technique. Thus, a use for logical sensors is evident in any sensor system which is composed of several sensors, and/or where sensor reconfiguration is desired.

The emergence of significant multi-sensor systems provides a major motivation for the development of logical (or symbolic) sensors. Monitoring highly automated factories or complex chemical processes requires the integration and analysis of diverse types of sensor measurements; e.g., it may be necessary to monitor temperature, pressure, reaction rates, etc. In many cases, fault tolerance is of vital concern; e.g., in a nuclear power

plant@cite[Nelson82]. Our work
has been done in the context of a robotic work station where the kinds of sensors involved include:
@begin(itemize)
cameras: an intensity array of the scene is produced,

tactile pads: local forces are sensed,

proximity sensors: the proximity of objects to a robot hand is sensed,

laser range finders: the distance to surface points of objects in the scene are produced, and

smart sensors: special algorithms implemented in hardware for detecting features such as edges.
@end(itemize)
Oftentimes, if the special hardware is not available, then some of these sensors may be implemented as a software/hardware combination which should be viewed as a distinct sensor and which ultimately may be replaced by special hardware.
Other examples of sophisticated sensor systems include
automatic target recognition (ATR) systems@cite(Bhanu83) and the Utah/MIT Dextrous Hand@cite(Jacobsen83). ATR systems integrate data from three (or more) sensors: microwave, FLIR, and LADAR.
The Utah/MIT Hand includes both internal and external (tactile) sensing systems.

Other principal motivations for logical sensor specification are:
@begin(itemize)
@u(benefits of data abstraction): the specification of a sensor is separated from its implementation. The multi-sensor system is then much more portable in that the specifications remain the same over a wide range of implementations. Moreover, alternative mechanisms can be specified to produce the same sensor information but perhaps with different precision or at different rates. Thus, several dimensions of sensor granularity can be defined. Further, the stress on modularity not only contributes to intellectual manageability@cite[Wirth79] but is also an essential component of the system's reconfigurable nature. The inherent hierarchical structuring of logical sensors further aids system development.

@u(availability of smart sensors): the lowering cost of hardware combined with developing methodologies for the transformation from high level algorithmic languages to silicon have made possible a system view in which hardware/software divisions are transparent. It is now possible to incorporate fairly complex algorithms directly into hardware. Thus, the substitution of hardware for software (and vice versa) should be transparent above the implementation level.
@end(itemize)

@section(Related Work)
The work most related, in a high level way, to logical sensor specification has been done in computer graphics. The need for some
device-independent interactive system has been widely recognized in the area of graphics, and the Graphical Kernel System (GKS) is now a Draft International Standard, and is under consideration as an American National Standard. The main idea behind GKS is to provide "a means whereby interactive graphics applications could be insulated from the peculiarities of the input devices of particular terminals, and thereby become portable"@cite(Rosenthal82).
This was accomplished by allowing only a restricted view
of an input device; the only aspect of an input device which could be viewed was the @u(type) of its output. Input devices so restricted are called @u(virtual input devices).

Criticisms of GKS have focused on the need for virtual devices to have visible aspects other than type alone. This led to the adoption of the @u(logical) device concept, which is a virtual device with an enlarged view whereby other details of importance are visible.

Logical sensors are also proposed as a means by which to insulate the user from the peculiarities of input devices, which in this case are (generally) physical sensors. Thus, for example, a sensor system could be designed around camera input, without regard to the kind of camera being used. However, in addition to providing insulation from the vagaries of physical devices, logical sensor specification is also a means to create and package "virtual" physical sensors. For example, the kind of data produced by a physical laser range finder sensor could also be produced by two cameras and a stereo program. This similarity of output result is more important to the user than the fact that one way of getting it is by using one physical device, and the other way is by using two physical devices and a program. Logical sensor specification allows the user to ignore such differences of how output is produced, and treat different means of obtaining equivalent data as logically the same.

Another related graphics interface system is SYNGRAPH@cite(Olsen83). This system automatically generates graphical user interfaces. The user expresses the desired interface in a modified BNF wherein a primitive input device must be declared so that a set of special features as well as output type are visible. A grammar-driven approach is favored because the syntactic description makes automated analysis of the interface possible.

The need for higher-level robotics languages has also been articulated by Donner@cite[Donner83] in his work on the OWL language. However, OWL is not a sensor specification language, but rather a simple programming language for describing concurrent processes to control a walking machine. More recently, there have been some encouraging results reported in the robotics literature. A systematic study of robotic sensor design for dynamic sensing has been undertaken by Beni et al@cite[Beni83]. Another research effort related to our work is the programming environment (called the Graphical Image Processing Language) under development as part of the IPON project (an advanced architecture for image processing) at the University of Pennsylvania@cite[Bajscy84a].

@section(Logical Sensors)

We have briefly touched on the role of logical sensors above. We now formally define logical sensors.

A @b(logical sensor) is defined in terms of four parts:

@begin(enumerate)

A @b(logical sensor name). This is used to uniquely identify the logical sensor.

A @b(characteristic output vector). This is basically a vector of types which serves as a description of the output vectors that will be produced by the logical sensor. Thus, the output of a logical sensor is a set (or stream) of vectors, each of which is of the type declared by that logical sensor's characteristic output vector. The type may be any standard type (e.g., real, integer), a user generated type, or a well-defined subrange of either. When an output vector is of the type declared by a characteristic output vector (i.e., the cross product of the vector element types), we say that the output vector is an "instantiation" of that characteristic output vector.

A @b(selector). The role of

the selector is to detect failure of an alternate and switch to a different alternate. If switching cannot be done, the selector reports failure of the logical sensor.

@b(Alternate subnets). This is a list of one or more alternate ways in which to obtain data with the same characteristic output vector. Hence, each alternate subnet is equivalent, with regard to type, to all other alternate subnets in the list, and can serve as backups in case of failure. @u(Each) alternate subnet in the list is itself composed of:

@begin(itemize)

A set of @b(input sources). Each element of the set must either be itself a logical sensor, or the empty set (null). Allowing null input permits @b(physical) sensors, which have only an associated program (the device driver), to be described as a logical sensor, thereby permitting uniformity of sensor treatment.

A @b(computation unit) over the input sources. Currently such computation units are software programs, but in the future, hardware units may also be used.
@end(itemize)

A @b[control command interpreter]. Each logical sensor has a set of control commands (specified by a grammar). The control command interpreter decodes these commands and produces the appropriate commands for the currently selected input logical sensors.
@end(enumerate)
Figure 1 gives a schematic view of logical sensors.
@Blankspace(3inches)
@center(@b[Figure 1]. The Logical Sensor Schema)
@begin(format)

@end(format)
A logical sensor can be viewed as a network composed of sub-networks which are themselves logical sensors.
Communication within a network is controlled via the flow of data from one sub-network to another. Hence, such networks are @u(data flow) networks.

The idea is that a logical sensor can specify either a device driver program which needs no other logical sensor input, but rather gets its input directly from the physical device and then formats it for output in a characteristic form, or a logical sensor can specify that the output of other logical sensors be routed to a certain program and the result packaged as indicated. This allows the user to create "packages" of methods which produce equivalent data, while ignoring the internal configurations of those "packages."
@subsection(Formal Aspects)
Having described how logical sensors are developed and operate, we now define a logical sensor to be a @b(network) composed of one or more sub-networks, where each sub-network is a logical sensor. The computation units of the logical sensor are the nodes of the network. Currently, the network
forms a rooted directed acyclic graph. The graph is rooted because, taken entirely, it forms a complete description of a single logical sensor (versus, for example, being a description of two logical sensors which share sub-networks). We also say that it is rooted because there exists a path between each sub-network and a computation unit of the final logical sensor. Logical sensors may not be defined in terms of themselves, that is, no recursion is allowed, and hence the graph is acyclic.

All communication within a network is accomplished via the flow of data from one sub-network to another. No explicit control mechanism, such

as the use of shared variables, alerts, interrupts, etc., is
allowed. The use of such control mechanisms would decrease the degree of
modularity and independent operation of sub-networks. Hence the networks
described by the logical sensor specification language are data flow
networks, and
have the following properties@cite(Keller78):
@begin(itemize)
A network is composed of independently, and possibly concurrently,
operating sub-networks.

A network, or some of its sub-networks, may communicate with its
environment via possibly-infinite input or output streams.

Sub-networks are modular.
@end(itemize)
Since the actual output produced by a sub-network may depend on things like
hardware failures (and because the output produced by the different
sub-nets of a logical sensor are only required to have the same type), the
sub-networks (and hence the network) are also indeterminate.
@subsection(Logical Sensor Specification Language)
It should be noted that there may be alternate input paths to a
particular sensor, and these correspond to the alternate subnets.
But even though there
may be more than one path through which a logical sensor produces data,
the output will be of the type declared
by the logical sensor's characteristic output vector.
With these points in mind, a language for describing the logical sensor
system can be formed. We give the syntax below.
@paragraph(Syntax)
@begin(verbatim)

(logical-sensor) ---> (logical-sensor-name)
(characteristic-output-vector)
(selector)
(alternate-subnet-list)
(control command interpreter)

(logical-sensor-name) ---> (identifier)

(characteristic-
output-vector) ---> (name-type-list)

(name-type-list) ---> (identifier):(type)
{;(name-type-list)}

(selector) ---> (acceptance-test-name)

(alternate-subnet-list) ---> (computation-unit-name) (input-list)
{(alternate-subnet-list)}*

(control command interpreter) ---> (identifier)

(acceptance-test-name) ---> (identifier)

(input-list) ---> (logical-sensor-list) | null

(logical-sensor-list) ---> (logical-sensor)
{(logical-sensor-list)}*

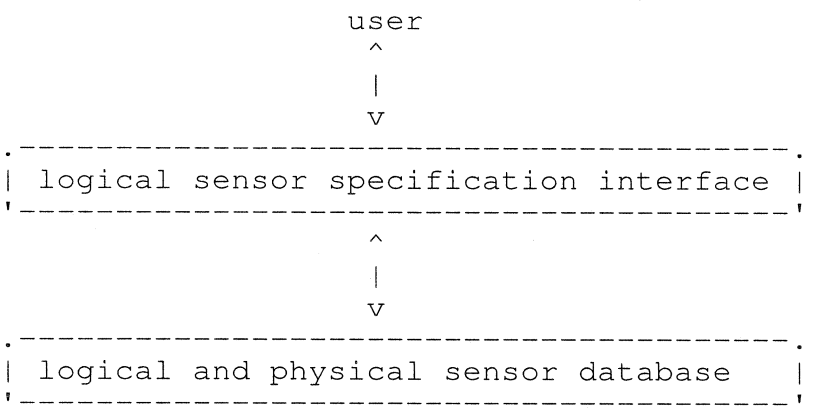
(computation-unit-name) ---> (identifier)

@end(verbatim)
@subsection(Implementation)
We currently have two implementations of the logical sensor specification

language running: a C version (called C-LSS) running under UNIX, and a functional language version (called FUN-LSS). The C version has been described elsewhere@cite[Henderson83d] and produces a shell script from the specification. We give details here of the functional language version.

FUN-LSS provides a logical sensor specification interface for the user and maintains a database of s-expressions which represents the logical sensor definitions (see Figure 2).

@begin(figure)
@begin(verbatim)



@center(@b[Figure 2]. The Logical Sensor System Interface)
@end(verbatim)
@end(figure)

The operations allowed on logical sensors include:

@begin(itemize)
@u(Create): a new logical sensor can be specified by giving all the necessary information and it is inserted in the database.

@u(Update): an existing logical sensor may have certain fields changed; in particular, alternative subnets can be added or deleted, program names and the corresponding sensor lists can be changed.

@u(Delete): a logical sensor can be deleted so long as no other logical sensor depends on it.

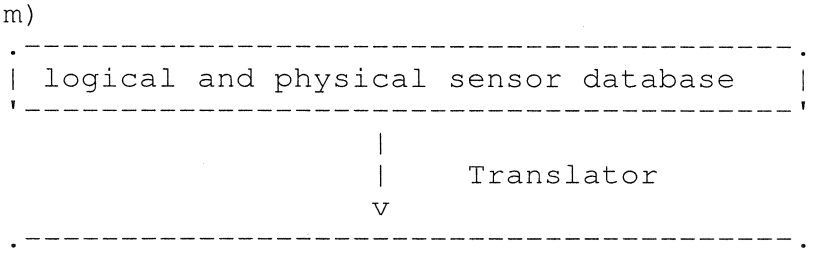
@u(Display): show all parts of a logical sensor or list all logical sensor names.

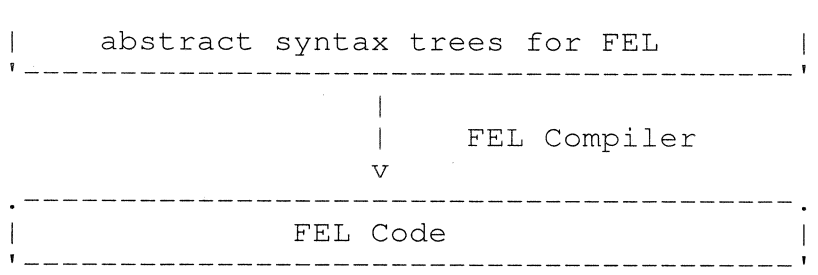
@u(Dependencies): show all logical sensor dependencies.
@end(itemize)

Once the logical sensors are specified, they are stored as s-expressions in the database. In order to actually execute the logical sensor specification, it is necessary to translate the database expressions into some executable form, e.g., to produce source for some target language, and then either interpret or compile and run that source.

Our approach is displayed in Figure 3.

@begin(figure)
@begin(verbatim)





@center(@b[Figure 3]. Steps to Obtain Executable Code)
@end(verbatim)
@end(figure)
We have written a translator which converts the s-expressions in the database into abstract syntax trees for a Function Equation Language (FEL)@cite[Keller82].
These are then passed to the FEL Compiler which produces a function graph which can then be evaluated, using a combination of graph reduction and dataflow strategies. More on these topics can be found elsewhere@cite[Shilcrat84]. In that paper we discuss a methodology for configuring systems of sensors using a functional language. The use of abstraction and of functional language features leads to a natural and simple approach to this problem. The features of a particular functional programming environment, Function Equation Language (FEL) running on the REDIFLOW simulator, are exploited to develop a scheme that avoids complicated issues of state restoration and switching protocols.
@Section(Fault Tolerance)
The Logical Sensor Specification Language has been designed in accordance with the view that languages should facilitate error determination and recovery.
As we have explained, a logical sensor has a selector which takes possibly many alternate subnets as input. The selector determines errors, and attempts recovery via switching to another alternate subnet. Each alternate subnet is an input source - computation unit pair. Selectors can detect failures which arise from either an input source or the computation unit. Thus, the selector together with the alternate subnets constitute a failure and substitution device, that is, a fault-tolerance mechanism, and @u(both) hardware and software fault tolerance can be achieved. This is particularly desirable in light of the fact that "fault tolerance does not necessarily require diagnosing the cause of the fault or @i(even deciding whether it arises from the hardware or software)" (emphasis added)@cite(Randell77). In a multi-sensor system, particularly where continuous operation is expected, trying to determine and correct the exact source of a failure may be prohibitively time-consuming.

Substitution choices may be based on either @u(replication) or @u(replacement). @u(Replication) means that exact duplicates of the failed component have been specified as alternate subnets. In @u(replacement) a different unit is substituted.
Replacement of software modules has long been recognized as necessary for software fault-tolerance, with the hope, as Randall states, that using a software module of independent design will facilitate coping "with the circumstances that caused the main component to fail"@cite(Randell77). We feel that replacement of physical sensors should be exploited both with Randall's point in view, and because extraneous considerations, such as cost, and spatial limitations as to placement ability are very likely to limit the number of purely back-up physical sensors which can be involved in a sensor system.

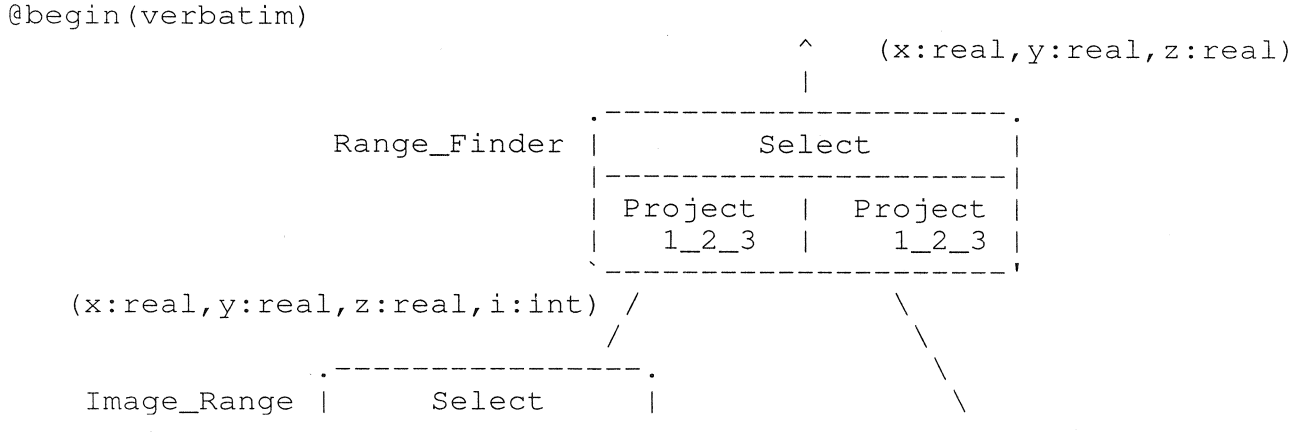
@subsection(Recovery Blocks)
The recovery block is a means of implementing

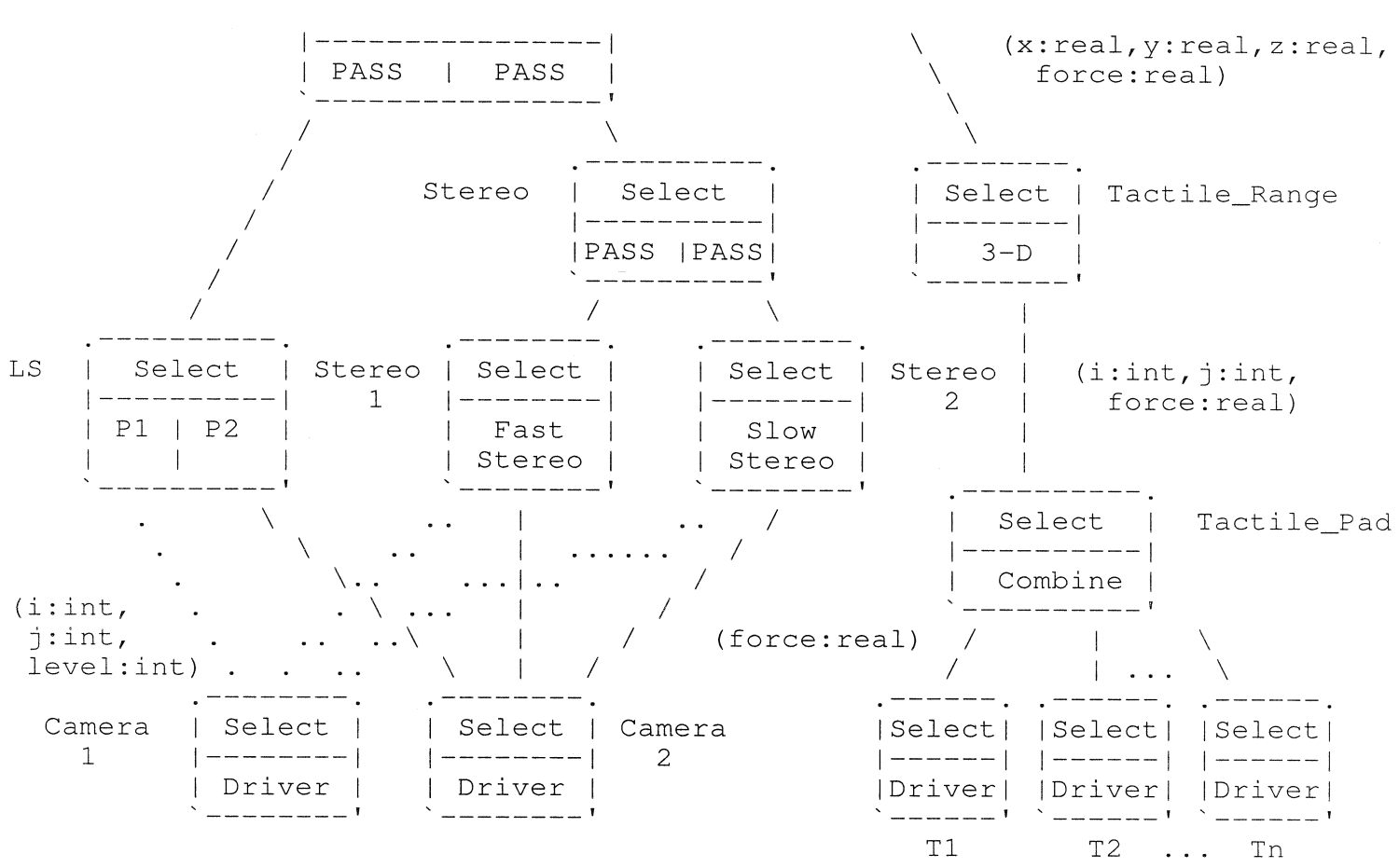
software fault tolerance@cite[Randell77].
A recovery block contains a series of alternates
which are to be executed in the order listed. Thus, the first in
the series of alternates is the @u(primary) alternate. An acceptance
test is used to ensure that the output produced by an
alternate is correct or acceptable. First the primary
alternate is executed, and its output scrutinized via the acceptance test.
If it passes, that block is exited, otherwise the next alternate is
tried, and so on. If no alternate passes, control switches to a new
recovery block if one (on the same or higher level) is available;
otherwise, an error results.

Similarly, a selector tries, in turn, each alternate subnet in the list,
and tests each one's output via an acceptance test.
However, while Randall's scheme requires the use of
complicated error recovery mechanisms (restoring the state, and so on), the
use of a data-flow model makes error-recovery relatively easy.
Furthermore, our user interface computes the dependency
relation between logical sensors@cite(Shilcrat84). This permits the system
to know which other sensors are possibly affected by the failure of a given
sensor.

The general difficulties relating to software acceptance
tests, such as how to devise them, how to make them simpler than the
software module being tested, and so on, remain. It is our
view that some acceptance tests will have to be designed by the user, and
that our goal is simply to accommodate the use of the test. Unlike
Randall, we envision the recovery block as a means for both hardware and
software fault-tolerance, and hence we also allow the user to specify
general hardware acceptance tests. Such tests may be based, for example,
on data link
control information, 2-way handshaking and other protocols.
It is important to note that a selector must be specified even if
there is only one subnet in a logical sensor's list of alternate subnets.
Without at least the minimal acceptance test of a "time-out," a logical
sensor could be placed on hold forever even when alternate ways to
obtain the necessary data could have been executed. Given the minimal
acceptance test, the selector will at least be able to signal failure to a
higher level selector which may then institute a recovery.
However, we also wish to devise special schemes for acceptance tests when
the basis for substitution is replacement. While users will often know
which logical sensors are functionally equivalent, it is
also likely that not all possible substitutions of
logical sensors will be considered. Thus, we are
interested in helping the user expand what is considered functionally
equivalent.
Such a tool could also be used to automatically generate
logical sensors.

We give an example logical sensor network in Figure 4.
@begin(fullpagefigure)
@begin(verbatim)





@center(@b[Figure 4]. Logical sensor network for Range_Finder.)
@end(verbatim)
@end(fullpagefigure)
This example shows how to obtain surface point data from possible alternate methods. The characteristic output vector of Range_Finder is (x:real,y:real,z:real) and is produced by selecting one of the two alternate subnets and
"projecting" the first three elements of their characteristic output vectors. The preferred subnet is composed of the logical sensor Image_Range. This logical sensor has two alternate subnets which both have the dummy computational unit PASS. PASS does not effect the type of the logical sensor. These alternatives will be selected in turn to produce the characteristic output vector (x:real,y:real,z:real,i:integer). If both alternates fail (whether due to hardware or software), the Image_Range sensor has failed. The Range_Finder then selects the second subnet to obtain the (x:real,y:real,z:real) information from the
Tactile_Range's characteristic output vector. If the Tactile_Range subsequently fails, then the Range_Finder fails. Each subnet uses this mechanism to provide fault tolerance.
@subsection(Ramifications of Fault-Tolerance Based on a Replacement Scheme)
Many difficult issues arise when fault tolerance is based on a replacement scheme.
Because the replacement scheme is implemented through the use of alternate subnets, the user can be sure that the @u(type) of output will remain constant, regardless of the particular source subnet. Ideally, however, we consider that a replacement based scheme is truly fault tolerant only if the effect of the replacement is within allowable limits, where the allowable limits are determined by the user. As a simple example, consider a sensor system of one camera, A, and a back-up camera, of another type, B. Suppose camera A has accuracy of @u(+) 0.01%, and camera B has accuracy of @u(+)0.04%. If the user has determined that the allowable limit on accuracy is @u(+)0.03%, then replacement of camera A by camera B will not yield what we call a truly fault tolerant system; if the allowable limit is @u(+)0.05%, the replacement does yield a

truly fault tolerant system, as it will if the user has determined that the system should run regardless of the degree of accuracy.

As mentioned above, determining functional equivalence may necessitate seeing more of a logical sensor than merely its type. This example illustrates this point in that we have isolated a need to know more about leaf logical sensors (physical sensors). However, we also mentioned that the above example was simplified. Let us now assume, in addition, that the user can use a variety of algorithms to obtain the desired final output. Suppose one of those algorithms incorporates interpolation techniques which could increase the degree of accuracy over camera B's input. In this case, the user may be able to use camera B and this algorithm as an alternate subnet and have a truly fault tolerant system, even if camera B's output is not itself within the allowable accuracy limit. Thus, when we consider a slightly more complex example, we see a general need for having features (beside type of output) of logical sensors visible, and a need to propagate such information through the system.

Feature propagation, together with allowable limit information, is needed for replacement based fault-tolerance schemes, and constitutes an acceptance test mechanism. In addition, such feature propagation has a good potential for use in automatic logical sensor system specification/optimization. For example, consider a workstation with several sensors. Once various logical sensors have been defined and stored, feature propagation can be used to configure new logical sensors with properties in specified ranges, or to determine the best (within the specified, perhaps weighted, parameters) logical sensor system. Thus, feature propagation is necessary for both fault tolerance and automatic generation of logical sensor systems, and it is our view that the basic scheme will be the same in either case.

@section(Features and Their Propagation)
Our view is that propagation of features will occur from the leaf nodes to the root of the network. In sensor systems, the leaf nodes will generally be physical sensors (with associated drivers). Thus, we first discuss the important features of physical sensors.
@subsection(Features of Physical Sensors)
Our goal here is to determine whether a set of generally applicable physical sensor features exists, and then to provide a database to support the propagation mechanism. In addition, it is possible for the user to extend the set of features. Currently, the system provides a small set of generally applicable features (see below).

All physical sensors convert physical properties or measurements to some alternative form, and hence are transducers. Some standard terms for use in considering transducer performance must be defined@cite(Wright83). We have selected a set of features defined by Wright which we feel are generally applicable to physical sensors.

@begin(itemize)
@u(Error) - the difference between the value of a variable indicated by the instrument and the true value at the input.

@u(Accuracy) - the relationship of the output to the true input within certain probability limits. Accuracy is a function of nonlinearities, hysteresis, temperature variation, and drift.

@u(Repeatability) - the closeness of agreement within a group of measurements at the same input conditions.

@u(Drift) - the change in output that may occur despite constant input conditions.

@u(Resolution) - the smallest change in input that will result in a significant change in transducer output.

@u(Hysteresis) - a measure of the effect of history on the transducer.

@u(Threshold) - the minimum change in input required to change the output from a zero indication. For digital systems this is the input required for 1 bit change in output.

@u(Range) - the maximum range of input variable over which the transducer can operate.
@end(itemize)

Based on this set of physical sensor characteristics, the next step in arriving at a characterization of logical sensors is to "compose" physical sensor feature information with computation unit feature information.
@subsection(Algorithm Features)
There are several difficult issues involved in choosing a scheme whereby features of algorithms can be "composed" with features of physical sensors such that the overall logical sensor may be classified. As Bhanu@cite(Bhanu83) has pointed out: "the design of the system should be such that each of its components makes maximum use of the input data characteristics and its goals are in conformity with the end result."

One issue to be resolved is how to represent features and feature composition. One approach is to record feature information and composition functions separately. Thus, it would be necessary to classify an algorithm as having a certain degree of accuracy, and, in addition provide an accuracy function which, given the accuracy of the physical sensor, produces the overall accuracy for the logical sensor which results from the compostion of the physical sensor and the algorithm. A major difficulty in resolving such issues is presented by the great variety of sensor systems, both actual and potential, and the varying level of awareness of such issues within different sensor user communities. For example, experienced users of certain types of sensors may have a fairly tight knowledge of when and why certain algorithms work well, whereas other user communities may be aware in only a vague way which algorithms work well under which circumstances. Indeed, even within a sensor user community, algorithm evaluation techniques may not be standardized, hence yielding a plethora of ways in which properties algorithms may be described. This problem is manifest in Bhanu's survey of the evaluation of automatic target recognition (ATR) algorithms@cite(Bhanu83).

The state of the art in algorithm evaluation techniques effects the choices made regarding the use of classifying physical sensors whether we wish to simply catalog information or maximize criteria. For example, if the user cannot provide information about the degree of resolution for the algorithms being used, then an overall logical sensor resolution figure cannot be determined, even if the resolution of all physical sensors is known. Also, if such is the case, then the system cannot be used to help the user maximize the degree of resolution of the final output.

On the other hand, there are some encouraging results reported in the literature; a systematic study of robotic sensor design for dynamic sensing has recently been undertaken by Beni et al@cite[Beni83], and more of that kind of work is required if we are to achieve comprehensive sensor systems.

@section(Automatic Logical Sensor Synthesis)

We are investigating ways in which to generate logical sensor systems automatically. We recognize that, considering the number of unanswered questions, we will not be able to establish a fully automatic logical sensor system, and therefore we propose to confine ourselves to an automatic logical sensor system of limited generality.

We now describe some techniques to allow for dynamic specification and allocation of logical sensors. Though the kinds of logical sensors which we consider represent only simple extensions to the existing logical sensor system, this type of work is a first step towards generally extensible logical sensor systems.

The goal here is to show how, given information about logical sensors which can be configured in the system, new logical sensors can be automatically defined.

@Subsection(Tupling Data)

Tupling data is a technique which can be used to automatically generate new logical sensors in a feature-based sensor system. In such systems, the logical sensors would be returning information about certain features found in the scene, such as number of edges, number of holes, temperature, metallic composition, and so on.

The user may then request that a new logical sensor be established by specifying the name for the new logical sensor, and giving the names of the input logical sensor(s). The output of the new logical sensor will be, simply, a set of tuples (one for each object in the scene), where the tuple is composed of the cartesian product of the features which were input from the source logical sensors. Thus, we are basically packaging together features of interest so that they will be in one output stream.

For example, suppose that the features "number of edges" and "number of holes" are sufficient to determine the presence of bolts. Then a logical sensor @u(bolt-detector) could be created by tupling the output of the logical sensors @u(edge-detector) and @u(hole-detector). It should be noted that we assume that the latter two logical sensors produce output of the form (object no., feature1, feature2, ... feature N). For the sake of simplicity, in this example we assume that logical sensor @u(edge-detector) produces output of the form (object no., number of edges) and logical sensor @u(hole-detector) produces output of the form (object no., number of holes). Logical sensor @u(bolt-detector) will match on object number, and produce tuples of the form (object no., number of edges, number of holes).

@subsection(Choosing Algorithms Based on Appropriateness/Reliability)

Our view is that a feature propagation mechanism is useful for both fault-tolerance checking and logical sensor optimization.

Some difficulties are involved in using the feature propagation mechanism in a logical sensor optimization system. From the optimization viewpoint, the task which we wish the logical sensor system to perform is not merely to produce output, but to produce output which is optimal. One difficulty is that what makes the output optimal may change from application to application, or from use to use. Hence, the logical sensor system should produce output of the specified type which is optimized according to the @u(user specified optimization criteria).

In light of the above discussed difficulties in developing a feature propagation mechanism, we

are considering optimization facilities which could also be used in the absence of a general feature propagation mechanism. Our goal is to help the user choose algorithms which maximize desired capabilities of a logical sensor system. Therefore, in addition to providing what may only constitute a catalog of physical sensor characteristics, we wish to establish a database of algorithms which can be searched to determine how to configure the optimal logical sensor system for the task at hand. Since, once again, we are forced to consider the level of information detail which the user can provide in setting up the database, we recognize that this database may or may not be part of a general feature propagation mechanism. In other words, if the user tell us only that a certain algorithm works well, for example, then this database will basically serve merely as an automatic cataloging device. On the other hand, if we can be provided with numerical estimates of certain parameters for each algorithm, and composing functions, the database can be used as part of a feature propagation mechanism. In the latter case, not only can we provide a much closer realization of the user's goal, but we may also be able to indicate which performance attributes cannot be met by any known configuration of physical sensors and algorithms; in such cases, the system may actually specify a new or the parameters on an algorithm which would make the demanded performance possible.

@subsection(Automatic Generation of Algorithm Feature Information)

Several approaches to the incorporation of algorithm feature information into a logical sensor specification system have been discussed. As an extension to this idea, we intend to investigate ways in which to use a logical sensor specification system to @u(generate) algorithm feature information. We are looking into the use of models for algorithm evaluation, together with a database of training data, that is, sample data to be used as a standard against which algorithms are evaluated. For the ATR (Automatic Target Recognition) systems, Bhanu states that the models for algorithm evaluation should be chosen such that each part of the system should be evaluated with respect to its own figures of merit but also against its effect on the overall classification (i.e., the overall goal of the system). In this view, statistical measures of an algorithm's performance such as edge point measures and structural measures, the ability of an algorithm to make maximal use of the specific characteristics of FLIR images, and the three general parameters which are used to determine the overall performance of an ATR system (probability of target detection, probability of classification, and false alarm per frame) must all be taken into account when evaluating an algorithm. In addition, these statistical, heuristic and parametric models are to be used in establishing the requirements of the database in terms of data collection and organization, with the end goal of generating databases of FLIR images which are increasingly representative of the real world. Thus, Bhanu envisions a training data base - algorithm data base interaction such that the original figures of merit for algorithms are refined, on the basis of sample data, to reflect ability to make maximal use of specific characteristics of particular physical sensor data towards the end of promoting the overall system performance. We agree with the philosophy that sensor systems should be viewed as the best source of information on to how to improve themselves, and intend to investigate the use of training databases, and possible training database - algorithm database interaction schemes.

@Section(An Example: A CAGD-Based Vision System)

Computer vision has been an active research area for over 20 years. In the early days, emphasis was on low level processing such as intensity and signal processing to perform edge detection@cite(ballard82,Rosenfeld76b). Systems were constructed which only operated in very

constrained environments or for very specific tasks@cite(barrow78, horn70,Witkin81). It was quickly recognized that higher level concepts of image
@i(understanding) were needed to successfully perform computer vision. More recently, models of objects and knowledge of the working environment have provided the basis for driving vision systems. This is known as model based vision. The pursuit of the fully automated assembly environment has fueled interest in model based computer vision and object manipulation.

The problem we are interested in solving is model based visual recognition and manipulation of objects in the automation environment. This involves building a 3-D model of the object, matching the sensed environment with the known world and locating objects. Not until the desired object is located and its orientation is known can a robot gripper or hand manipulate it.

Our goal is to develop a system which will work in the environment of the automated assembly process. This is not intended to provide a general model for the human visual process but rather a solution to the problem of visual recognition and manipulation in a well-known domain. The constraint we are imposing is one which limits the necessity of modeling the entire world. Rather, the known world to us is that of the automated environment in which this system is intended to operate.

In order to recognize an object, vision algorithms require a model of the object. Different shape representations
usually necessitate different recognition techniques. Currently, computer vision systems are being developed which permit the use of multiple representations to describe a single object. The representations can be constrained by the class of shapes, the class of algorithms or by other means. In any case, some model must initially be constructed in order for the system to operate.

Likewise, Computer Aided Geometric Design (CAGD) systems allow the designer to model objects to be produced. These systems may include facilities for modeling certain classes of objects, such as sculptured surfaces (primarily used in free form design) or combination of primitives (primarily used in mechanical design). Just as in machine vision, the representation of shape is a central problem.

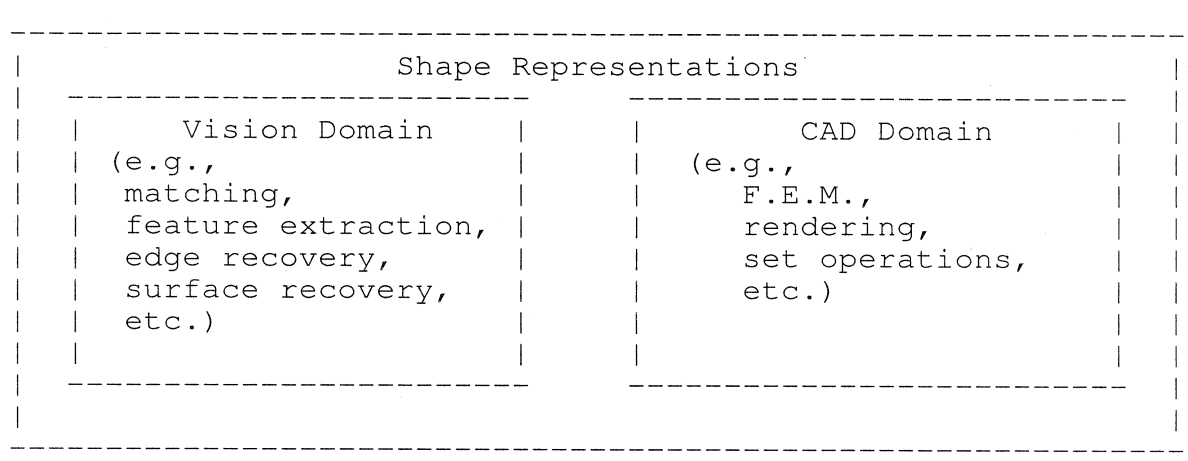
Even though these similarities exist, there is a dichotomy in computer vision and CAGD. A model is created by the designer and is stored in the CAGD database. In order for a visual inspection station to operate, the user must design the appropriate vision model, constrained
possibly by the class of shapes or the algorithm chosen, for the system. Thus, in current systems, the same object would have to be modeled more than once. Although shape modeling is performed in both disciplines, the models are not related except that they may represent the same object. More recently, research has focused on the problem of linking the two disciplines@cite(Castore84, Kuan83).

Simply stated, our proposed approach is to provide an integrated environment in which the CAGD model can be used to generate appropriate recognition and manipulation strategies. A major aspect of this work is the successful development of a prototype system combining design, vision analysis and manipulation. The key to our approach is the notion of @u(specialization). Rather than use a general weak method for visual recognition, we use logical sensor specifications to synthesize specialized, finely-tuned recognition "packages" which are specific to a particular shape or feature.

@Subsection(Shape Analysis)

In 3-D shape analysis, we have a 3-D model of known objects and a way of matching objects in a scene with the model. Shape representations and the algorithms which operate on these representations are intimately tied

together. Shape representations are chosen to solve specific problems. A representation which is good for CAGD might not be as effective (where effective might mean efficient or successful) when used in computer vision. Figure 5 illustrates some aspects of this.
@begin(Figure)
@begin(verbatim)

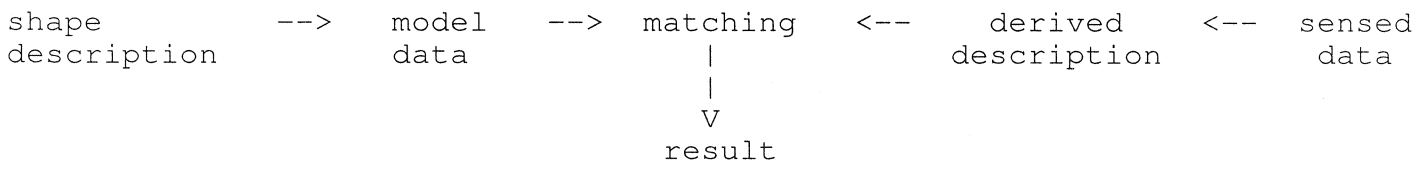


@end(verbatim)
@center(@b[Figure 5]. CAGD Representations vs. Computer Vision Representations)

@end(Figure)
Representations which are used in CAGD support some basic operations which are fundamental for graphics or design applications. They should provide an efficient means for rendering, finite element analysis, set operations, etc.

Although the objects modeled may be the same, representations appropriate for the above processes do not necessarily best serve computer vision's needs. Representations for computer vision need to support feature models, active sensing, edge recovery and relations, and explicit surfaces. Furthermore, a representation which might be efficient and concise for one vision application might not be appropriate for another vision application. For example, a representation which is efficient for feature models isn't likely to be efficient when used for edge recovery and relations.

There are many different recognition strategies which can be used to locate an object in a scene. Recognition algorithms are generally developed to operate on a particular representation. Marr has promoted the concept of the 2 1/2-D sketch representation and methods which operate on it@cite(Marr75, Marr77, Nishihara81). Other researchers have developed the syntactic approach which involves shape parsing and uses shape grammars as the representation@cite(Davis79, Davis81, Henderson79, Henderson81, Henderson81d, Henderson83c1, Henderson83c2, Lin84). Another approach is based on features of the shape where a feature vector encoding the shape is extracted and matching algorithms operate on these@cite(Henderson81e, Henderson82c, Ishii76). Others perform the matching operation using surface patches@cite(Aggarwal,bhanu84). Another popular method for shape recognition is graph matching@cite(pavlidis77). Figure 6 shows the general schema for these shape analysis methods.
@begin(Figure)
@begin(verbatim)



@center(@b[Figure 6]. Schema for Shape Analysis)

@end(verbatim)
@end(Figure)

A major aspect of choosing a representation method is mapping the representation to a matching strategy for object recognition. The availability of matching paradigms might restrict the possible choices of either representation methods or implementation schemes.

The system we are developing integrates the CAGD design system with the robotic workcell. The system contains knowledge of recognition strategies, shape representations, available sensors, and manipulation strategies. It uses this knowledge to guide the vision system and robot in the process of recognizing, locating and manipulating objects in the workcell environment.

The key issues in automatic generation of recognition strategies are:
@begin(enumerate)
generating an object model in the chosen representation from a CAGD model base, and

matching the correct shape representation with known recognition algorithms and sensors available.
@end(enumerate)
The most difficult of these these two is selecting the appropriate shape representation for the vision model. Once the shape representation is chosen, the object model for the vision system must be generated. While this may not be straightforward, algorithms can be developed which can perform this transformation. The problem of selecting a representation for the vision model is constrained by several factors. One is the availability of recognition algorithms. If we consider the available algorithms to be stored in a library, the selection can be constrained by this library. The trivial case of selecting the correct representation occurs when the recognition library of known strategies is limited to one representation. This can be considered the trivial case, even though the transformation from the CAGD model base may be nontrivial, since the selection of the shape representation is dictated by the singleton library of recognition schemes. Similarly, knowledge of the sensors available in the robotic workcell will further constrain the recognition procedure. These too can be thought of as being a library of available sensors.

The process is further complicated by the existence of CAGD models composed of multiple representations. For each complete CAGD model, there might possibly be several forms of representations contributing to the final result. If we think of the CAGD model as forming a tree of representations whose leaves are homogeneous models, we can match each of the shapes represented by these homogeneous models with some shape matching algorithm available to us in the library. Figure 7 demonstrates this idea. Consider a CAGD model to be made up of multiple structures, @B(S@-(i)), each of which might possibly be in a different representational form. For each of the @B(S@-(i))'s, the system will select an appropriate algorithm and sensor type to perform the matching the workcell. This will constrain the type vision model, @B(M@-(i)), to be used.
@Begin(Figure)
@begin(verbatim)

```
          /   \
        S2     S3
```

@center(@b[Figure 7.] Relation of CAGD Models to Vision Models)

@end(verbatim)
@end(Figure)

Once the representation strategy is determined, the transformation from the CAGD representation to the recognition representation must be performed. Knowledge of this transformation is encoded along with knowledge of existing recognition algorithms. Thus, the method for the transformation can be explicit in the system. For example, if the recognition strategy uses generalized sweep, the model built from the CAGD model base would be in the form of sections of the generalized cylinder. Should planar or quadric patches be selected, the representation for recognition would be a graph structure of relations between the patches. If feature vectors are the chosen method for recognition, the features can be extracted directly from the CAGD model or the CAGD system might first produce an image of one view of the object then the features could be extracted by the same algorithm which processes the sensed data.

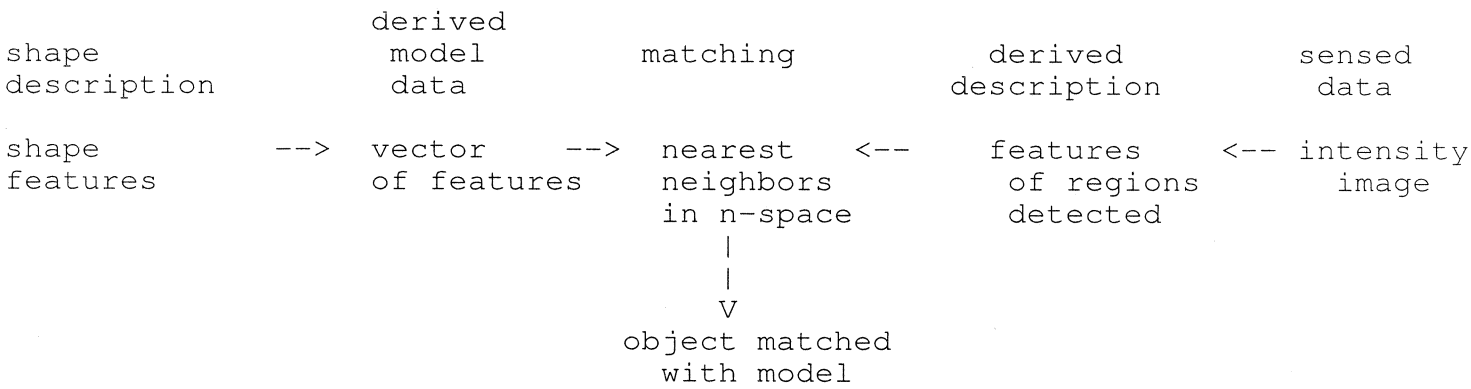
An extreme method for generation of recognition strategies is parameterization. This is extreme because the user is required to @I(fill in the blanks) for the sensors and algorithms for the particular object, or class of objects, modeled. Obviously, a more automated system is desired for this task. Drawing from our experience with Logical Sensor Specification in the MKS system, a possible alternative is to combine several algorithms and sensors to form a specialized @I(object finder)@cite(hansen83, henderson83d, henderson84c, henderson84f, henderson85,hende
rson85a, henderson85c, henderson85e, henderson85h).
The methodology provides a means for abstracting the specification of a sensor from its implementation along with providing transparency of hardware and software above the implementation level. Another alternative is to embed knowledge of the algorithms and sensors in the system and provide a rule base for the decision process. This would require a complex expert system (see, for example a description of a preliminary system@cite(henderson84e)). In either case, the system would be composed of multiple sensors and recognition methods.
@Subsection(2-D Binary Vision)
To demonstrate the concepts proposed, let us consider the design of an object, generation of the corresponding 2-D vision models and model based recognition of the object. We provide the functionality of an existing vision system currently in the market place. For this demonstration, we do not actually mill the piece even though the capabilities exist for the class of objects involved. Furthermore, we do not consider manipulation in this example but concentrate on the vision aspects instead. It is stressed that this is not intended to be a general solution to the problem but is rather an illustration of how a CAGD driven vision system might operate and indicative of the functionality of currently available commercial vision systems.

The vision system we emulate is an industrial type 2-D feature-based system. Such systems use features to distinguish objects. These features include area, perimeter, and several moments which are invariant to translation, rotation and scale. Images are obtained via a digitization process using a TV camera. The gray scale image is transformed into a binary image for subsequent processing. The recognition phase of the system is performed using a nearest neighbor classifier operating on a user selectable subset of the features.

Features vectors are representative of both currently available commercial systems and on going research efforts@cite(Bolles82). In feature vector

matching, the model is a set of features of the particular object in vector form. When a scene is analyzed, all objects are segmented into distinct regions and the pertinent features are extracted. These are then used to recognize known objects usually with a nearest neighbors method in n-dimensional space. Figure 8 is demonstrates how this method is applied.

@begin(Figure)
@begin(verbatim)



@center(@b[Figure 8.] Schema for Feature Vector Matching Paradigm)

@end(verbatim)
@end(Figure)

These systems obtain an object model by using the system to perform feature extraction on a physical example of the object to be recognized (sometimes called @I(training) the system).
As a result, when this system is used in an automated environment, the link between the CAGD model and the vision model would be the physical piece itself rather than the CAGD model.
Although this system seems rather elementary in its capabilities, it is representative of current products on the market. Indeed, most commercial systems sell for around $30,000.
@Subsection(Synthesis of Feature-Bases Object Detectors)

@Section(Conclusion)
We have described Logical Sensor Specifications as a framework facilitating efficient and coherent treatment of information provided in multi-sensor systems. In addition to the issues raised when considering the language implementation itself, various extensions have been suggested. In particular, we have implemented:
@begin(itemize)

A Logical Sensor Specification Language compiler.

General fault-tolerance features such as:
@begin(enumerate)
A mechanism for detecting sensor failure.

A technique by which switching to an alternate subnet is accomplished.

A method for determining when a sensor failure dictates top-level sensor failure.
@end(enumerate)

A database of physical sensors.

Automatic generation of tupling/merging logical sensors.

@end(itemize)

In addition, we are currently investigating:

@begin(itemize)
Formal semantics for the Logical Sensor Specification Language.

Features and feature propagation, in particular, how to arrive at a classification scheme for algorithm features and composing functions.

The establishment of an algorithm database for at least optimization purposes.

Inference schemes by which to determine a need for new physical sensors.

Training databases, and training database - algorithm database interaction schemes.

The automatic synthesis of logical sensor specifications.
@end(itemize)