@Make(Article)
@Device(PostScript)
@Style(Spacing 1.5, LineWidth 15.8cm, BottomMargin 1.3cm)
@LibraryFile[accent]
@begin(format)

# Multisensor Knowledge Systems

Thomas C. Henderson and Chuck Hansen
Department of Computer Science
The University of Utah

@b[Abstract]

@end(format)
We describe an approach which facilitates and makes explicit the organization of the knowledge necessary to map multisensor system requirements onto an appropriate assembly of algorithms, processors, sensors, and actuators.
We have previously introduced the Multisensor Kernel System and Logical Sensor Specifications as a
means for high-level specification of multisensor systems.
The main goals of such a characterization are:
to develop a coherent treatment of multisensor information,
to allow system reconfiguration for both fault tolerance and dynamic response to environmental conditions, and
to permit the explicit description of control.

In this paper we show how Logical Sensors can be incorporated into an object-based approach to the organization of multisensor systems. In particular, we discuss:
@begin(itemize)
a multisensor knowledge base,

a sensor specification scheme, and

a multisensor simulation environment.
@end(itemize)
We give example applications of the system to CAD-based vision@foot(This work was

supported in part by NSF Grants MCS-8221750, DCR-8506393, and DMC-8502115.)
.
@Blankspace(1.5cm)
@u(Introduction)
@Blankspace(1cm)
The rapid design of embedded electromechanical systems is crucial to success in manufacturing and defense applications. In order to achieve such a goal, it
is necessary to develop design environments for the specification, simulation, construction and validation of multisensor systems. Designing and prototyping such complex systems involves integrating mechanical parts, software, electronic hardware, sensors and actuators. Design of each of these kinds of
components requires appropriate insight and knowledge. This in turn has given rise to special computer-based design tools in each of these domains. Such Computer Aided Design (CAD) systems have greatly amplified the power and range of the human designer. To date, however, it is still extremely difficult to address overall system issues concerning

how the components fit together, and how the complete system will perform.

It is crucial to develop a design environment in which these multiple facets of system design can take place in a coordinated way such that the description of one component can be easily interfaced to another component, even when they are radically different kinds of things (e.g., a control algorithm, a mechanical linkage and an actuator). The designer should have the freedom to try out ideas at different levels of detail; i.e., from the level of a sketch to a fully detailed design. The Multisensor Knowledge System provides part of the solution to developing such an environment.

Logical Sensor Specifications (LSS) were developed previously as a method to permit an
implementation independent description of the required sensors and algorithms in a multisensor system. Figure 1 gives a pictorial description of the basic unit: a @i(logical sensor).
@begin(Figure)
@Blankspace(4.5inches)
@begin(format)

@center(@b[Figure 1.] Logical Sensor Specification Building Block:
The Logical Sensor)

@end(format)
@end(Figure)
Sensor data flows up through the currently executing program (one of program@-[1] to program@-[n]) whose output is characterized by the @i(characteristic output vector). Control commands are accepted by the @i(control command interpreter) which then issues the appropriate control commands to the Logical Sensors currently providing input to the selected program. The programs 1 through n provide alternative ways of producing the same characteristic output vector for the logical sensor. The role of the @i(selector) is to monitor the data produced by the currently selected program and the control commands. If failure of the program or a lower level input logical sensor is detected, the selector must undertake the appropriate error recovery mechanism and choose an alternative method (if possible) to produce the characteristic output vector. In addition, the selector must determine if the control commands require the execution of a different program to compute the characteristic output vector (i.e., whether dynamic reconfiguration is necessary).

Logical Sensor Specifications are useful then for any system composed of several sensors, where sensor reconfiguration is required, or where sensors must be actively controlled. The principle motivations for Logical Sensor Specifications are the emergence of significant multisensor and dynamically controlled systems, the benefits of data abstraction, and the availability of smart sensors.

In previous papers we have explored several issues of multisensor integration in the context of Logical Sensor Specifications:
@begin(itemize)
fault tolerance (Henderson 1984),

functional (or applicative) style programming (Shilcrat 1984a),

features and their propagation through a network (Shilcrat 1984b),

the specification of distributed sensing and
control (Henderson 1985a, Henderson 1985b),

the automatic synthesis of Logical Sensor Specifications for CAD/CAM applications (Henderson 1986a, Henderson 1986b].
@end(itemize)

Related work includes that of Albus (Albus 1981) on hierarchical control, Bajcsy et al. (Bajcsy 1984) on the Graphical Image Processing Language, Overton (Overton 1986) on
schemas, and Chiu (Chiu 1986) on functional language and multiprocessor implementations. For an overview of multisensor integration, see Mitiche and Aggarwal (Mitiche 1986), and for a recent review of multisensor integration research, see (Henderson 1987).

In exploring these issues, we have found that the specification of multisensor systems involves more than just sensor features. It is true that knowledge must be available concerning sensors, but it is essential to also be able to describe algorithms which use the sensor data and the hardware on which they are executed. In the rest of the paper, we describe the components of an object-based approach to developing a knowledge system to support these requirements.
@Blankspace(1.5cm)
@u(Objects and Methods)
@Blankspace(1cm)
Several distinct programming styles have been developed over the last few years, including:
@begin(itemize)
@b[applicative-style programming],

@b[control-based programming],

@b[logic programming], and

@b[object-based programming].
@end(itemize)
Applicative style programming exploits function application as its main operation and regulates quite strongly the use of
side-effects (Henderson 1980).
Historically, however, control-based programming has been the most extensively used paradigm, and focuses on the flow of control in a program. Logic programming is based on logical inference and requires the definition of the formal relations and objects which occur in a problem and the assertion of what relations are true in the solution.
On the other hand, many current systems are being developed which
are based on the notion of objects; this style emphasizes data abstraction combined with message passing (Booch 1983, Organick 1983).

In the control-based style a program is viewed as a controlled sequence of actions on its total set of data structures. As the complexity of a system grows, it is hard to keep a clear picture of the entire sequence of actions that make up the program. This leads to the chunking of sequences into subprograms, and this is almost exclusively done for control purposes. But data structures are not decomposed into independent entities. In fact, most global data structures are shared by all subroutines.

On the other hand, the object-based style takes the view that the major concern of programming
is essentially the definition, creation, manipulation and interaction of objects; that is, a set of independent and well-defined data structures. In particular, a single data structure (or instance) is associated with a fixed set of subprograms (methods), and those subprograms are the only operations defined on that object.

Such a use of data abstraction leads to design simplification which in turn makes the program more understandable, correct, and reliable. In addition, flexibility and portability are enhanced since details of objects (i.e., their representations) are hidden and can be implemented in other ways without changing the external behavior of the object.

For our purposes, an object consists, essentially, of three parts:
@begin(enumerate)
@u(unique name): this name must be distinguished from all other names in both time and space,

@u(type): an object is an instance of a type which defines the valid set of operations and which details the nature of the resource represented, and

@u(representation): the representation contains the information content associated with an object. This may include private data structures, references to other objects, etc.
@end(enumerate)

Thus, an object is a structure with internal state (perhaps called @i(slots) and comprised of name/value relationships) accessed through functions (also called @i[methods]) defined in association with the object. This approach makes management schemes simpler and fewer, easier to implement and use; in addition, individual resources are easier to specify, create (allocate), destroy (deallocate), manipulate and protect from misuse.

It has been effectively argued many times that object-based programming is well-suited to embedded systems processing requirements. In particular, the application of this methodology to the specification of sensor systems helps to directly describe most of the important aspects of such systems:
@begin(itemize)
parallel processing,

real-time control,

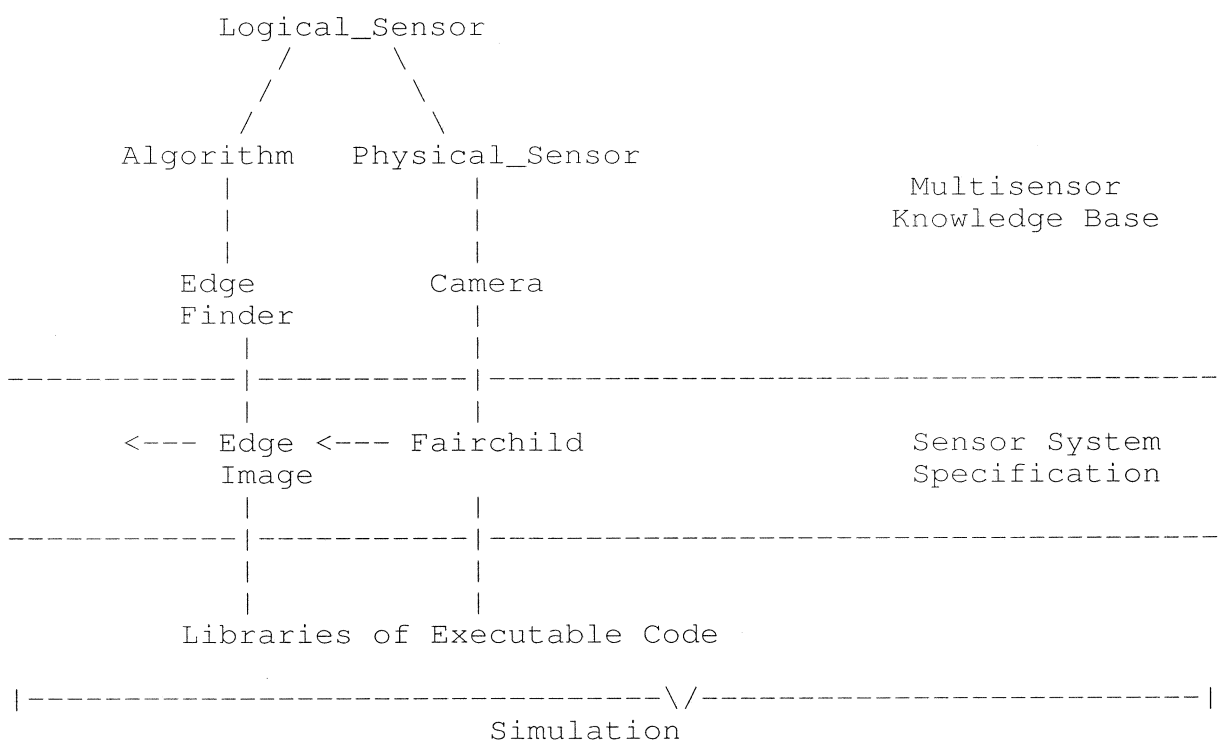
exception handling, and

unique I/O control.
@end(itemize)
Sensors typically require such operations as: enabling/disabling, limit setting, status checking, and periodic logging of state.
That is, sensor systems must respond to out-of-limit readings and issue alarms, detect faulty sensors, and recover from failure, and these functions can be implemented in a straightforward way.
@Blankspace(1.5cm)
@u(Multisensor Knowledge Systems)
@Blankspace(1cm)
Much of our previous work on multisensor systems has concentrated on the specification of such systems and reasoning about their properties. It is necessary to be able to describe both the parameters and characteristics of individual components of multisensor systems, and to be able to deduce global properties of complete systems. Although it may be possible to deduce such properties (especially static properties like complexity, data type coercion, etc.), we believe that many interesting properties can only be determined by simulating the operation of the complete system.

Thus, we seek a representation that supports:
@begin(enumerate)
@b[multisensor system specification]: this describes the components and interconnection scheme of the particular system being designed,

@b[sensor, algorithm, processor and actuator knowledge representation]: this structures information about
sensor characteristics (e.g., accuracy, hysteresis, dynamic range, etc.), algorithms (e.g., space and time complexity, amenity to parallel computation, stability, etc.) processors (e.g., cycle times, memory limits, address space, power requirements, etc.), and actuators (e.g., actuation principle, power requirements, etc.), and

@b[multisensor system simulation]: this permits one to monitor important parameters and to evaluate system performance.
@end(enumerate)
Figure 2 shows the organization of the three capabilities within an object-oriented context.
@begin(Figure)
@begin(comment)
@begin(verbatim)

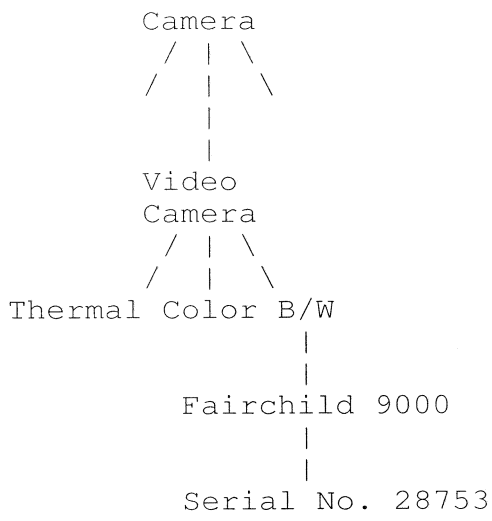



@end(verbatim)
@end(comment)
@Blankspace(5inches)
@center(@b[Figure 2]. Multisensor Knowledge System)
@end(Figure)
In the following subsections, we describe the @b[M]ultisensor @b[K]nowledge @b[S]ystem (@b[MKS]), an object-based approach to providing a unified answer to these three capabilities.
@Blankspace(1.5cm)
@u(The Multisensor Knowledge Base)
@Blankspace(1cm)
The multisensor knowledge base serves two main purposes:
@begin(enumerate)
to describe the properties of the system components (e.g., sensors, algorithms, actuators and processors), and

to provide class descriptions for the actual devices which are interconnected in any particular logical sensor specification.
@end(enumerate)
That is, the knowledge base must describe not only generic sensors (e.g., cameras), but specific sensors (e.g., Fairchild 9000, Serial No. 28753). It is then possible to reason about sensor systems at several levels. Moreover, it is possible that two distinct specifications require some of the same physical sensors. In such a case, it is the responsibility of the execution environment to resolve resource allocation conflicts.

We have chosen a frame-like knowledge representation. Frames relate very naturally to object-based descriptions, and, in fact, can be viewed as a class of restricted objects. It is straightforward to provide hierarchical descriptions of system components. For example, Figure 3 shows the

@i(CCD Camera) hierarchy.
@begin(Figure)
@begin(comment)
@begin(verbatim)




@end(verbatim)
@end(comment)
@Blankspace(5inches)
@center(@b[Figure 3]. Organization of Camera Knowledge)

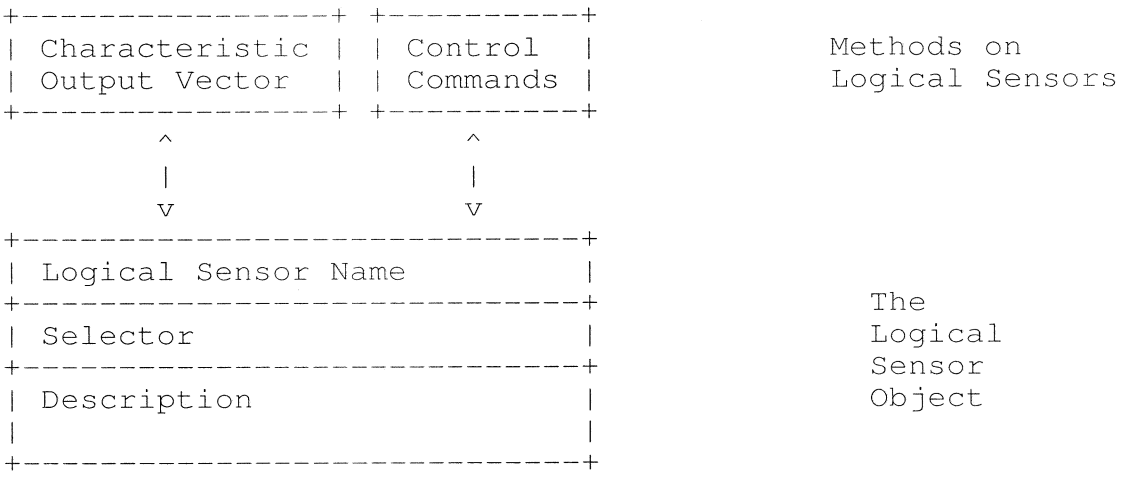
@end(Figure)
The @i(CCD Camera) frame has two slots: element spacing and aspect ratio. These slots are specific to CCD cameras and as such do not appear as slots for @i(2-D cameras). These latter have slots for scanning format, scan timing, resolution, output signal, and operating conditions. These slots are inherited by any instance of CCD camera. One level up, we find a frame for @i(Vision) sensors. This frame has specific slots for the spectral band and for the output type (e.g., 2-D byte array, multi-band, etc.). At the highest level of the hierarchy is the @i(Sensor) frame which has a slot for the physics of operation. This slot is used by any particular sensor to allow for an explanation of the physics behind the workings of the sensor. In this way, if reasoning is required about the sensor, it is possible to look in this slot for information.
As can be seen, knowledge is organized such that there are more specific details lower in the hierarchy.

Note that frames are themselves implemented as objects. Thus, actual devices are instances of some class of objects. This is very concise and conveniently exploits the similarities of frames and objects.

In previous work, we have described a set of generally applicable physical sensor features (Henderson 1984b). The manner in which physical sensors convert physical properties to some alternative form, i.e., their transducer performance, can be characterized by:
error,
accuracy,
repeatability,
drift,
resolution,
hysteresis,
threshold,
and range.
These properties can be encoded in the appropriate slots in the frames describing the sensor.
@Blankspace(1.5cm)

@u(Sensor Specification)
@Blankspace(1cm)
An object-based style of programming requires that the logical sensor of Figure 1 be re-described in terms of objects and methods. We shall next give the general flavor of this style, but it must be remembered that any particular sensor is actually an instance of some object class, and, in fact, inherits properties from many levels up.

Each logical sensor is completely specified as shown in Figure 4.
@begin(Figure)
@begin(comment)
@begin(verbatim)




@end(verbatim)
@end(comment)
@Blankspace(4inches)
@center(@b[Figure 4]. The Logical Sensor Object and Methods)
@end(Figure)
Thus, in order to get data from a logical sensor, the @i(characteristic output vector method) must be invoked. Likewise, to issue control commands to the sensor (e.g., camera pan and tilt parameters), the @i(control commands method) must be used. The role of the @i(selector) is still the same as in previous logical sensor implementations, however, it now, in essence, is invoked to produce the characteristic output vector.

Such a representation makes it very easy to design sensor systems. Moreover, such specifications allow for replacement of sensors and dynamic reconfiguration by simply having the @i(selector) send messages to different objects. Given current object-based programming technology, such systems can be rapidly developed and permit dynamic typechecking (on objects).

Figure 5 shows the Multisensor Knowledge Base, and below the dashed line, a set of particular instances of various algorithms, sensors, etc. (drawn as circles).
@begin(Figure)
@begin(format)
@Blankspace(3.5inches)

@center(@b[Figure 5]. Logical Sensor Specification Using Object Instances)

@end(format)
@end(Figure)
A logical sensor specification (indicated as a blocked in subset of the circles) defines a grouping of algorithms, sensors, etc. This newly created logical sensor is an instance of the @i(logical sensor object) and can be sent messages. As mentioned above, there are two methods defined on

logical sensors: the @i(characteristic output vector method) and the @i(control commands method). Thus, any logical sensor can be defined recursively in terms of other logical sensors (including itself).

Currently, our main interest is in the automatic synthesis of logical sensor specifications. Given a CAD model of an object, we would like to synthesize a specific, tailor-made system to inspect, recognize, locate or manipulate the object.
Note that the synthesis of a logical sensor specification consists, for the most part, of interconnecting instances of sensors and algorithms to perform the task. This is done by writing the selector to invoke methods on other logical sensors. Given certain constrained problems, most notably the CAD/CAM environment, such a synthesis is possible.
@Blankspace(1.5cm)
@u(The Simulation of Multisensor Systems)
@Blankspace(1cm)
Effective simulation plays an important role in successful system development. A key requirement is the support for hierarchical specification of the system and the ability to perform stepwise refinement of the system. In addition, it is necessary to be able to efficiently emulate realtime software that will eventually be embedded in the system. Finally, it would be quite useful to be able to embed physical components in the simulator in order to monitor the system's operation.

An object-oriented simulation methodology is well-suited to satisfy these goals. The multisensor system, that is, the system being modeled, consists of a collection of interacting physical processes. Each such process is modeled in the simulator by an object, i.e., an independent process. Interactions among physical processes are modeled through messages exchanged among the corresponding objects.

This general paradigm is currently supported in the SIMON simulator developed by Fujimoto (Fujimoto 1985, Swope 1986). A toolkit approach is used in which
the simulator is viewed as a collection of mechanisms from which application specific simulation environments are constructed. We are currently exploring the simulation of multisensor systems in the SIMON environment. Simulation can be accomplished by substituting simulation libraries for the run time libraries.

A crucial aspect of the simulation is the ability to execute specific algorithms on specific hardware. SIMON permits such a direct execution technique in which application programs are executed directly on a host processor rather than through a software interpreter. Performance information is obtained through the automatic insertion of probes and timing software into the program at compile time. These probes perform whatever runtime analysis is required to accurately estimate execution time of basic blocks of code. A prototype implementation using this technique has been developed modeling the MC68010 and 68020 microprocessors. Initial data indicate that application programs may be emulated one to two orders of magnitude more efficiently over traditional register transfer level simulation, while highly accurate performance estimates can still be obtained.
@Blankspace(1.5cm)
@u(An Example Application: CAD-Based 2-D Vision)
@Blankspace(1cm)
A simple example which demonstrates some of the power of the Multisensor Knowledge System approach is that of CAD-Based 2-D Vision. The goal is to automate visual inspection, recognition and localization of parts using pattern recognition techniques on features extracted from binary images. Figure 6 shows the scheme pictorially.
@begin(Figure)
@Blankspace(5inches)

@center(@b[Figure 6]. Synthesis of Part Detector)
@end(Figure)

The Multisensor Knowledge System stores knowledge about the algorithms, sensors, processors, etc. This knowledge is used by application specific rules. The systems to be synthesized here require that a model be created for the part to be inspected, and that a robust and (perhaps) independent set of features be chosen along with an appropriate distance metric.

The left side of the figure shows the offline training component. The new part is designed using a Computer Aided Geometric Design system. A set of images are rendered by the CAGD system giving a sample of various views of the part in different positions, orientations, and scales. These serve as a training set to the Multisensor Knowledge System.

A set of rules (or productions) performs an analysis of the views of the part to select a subset of the total set of possible features. Features are used if they are robust, independent and reliable. Once these features have been chosen, a new logical sensor object is created whose only function is to recognize the given part based on an analysis of the selected features. The part detector is then linked into a particular application (e.g., an inspection task at a specific workcell) by sending a message to the appropriate camera.

As a specific example, consider the object shown in Figure 7.
@begin(figure)
@begin(format)
@Blankspace(7inches)

@center(@b[Figure 7]. The Designed Piece)
@end(format)
@end(figure)
It was designed with Alpha_1, an experimental solid modeling system developed at the University of Utah. For the past few years the Computer Aided Geometric Design group has been involved in a concerted effort to
build this advanced modeler. Alpha_1 incorporates
sculptured surfaces and embodies many theoretical and algorithmic advances. It allows in a
single system both high-quality computer graphics and freeform surface representation and design. It uses a rational polynomial spline representation of arbitrary degree to represent the basic shapes of the models. The rational B-spline includes all spline polynomial representations for which the denominator is trivial. Nontrivial denominators lead to all conic curves. Alpha_1 uses the Oslo algorithm for computing discrete B-splines. Subdivision, effected by the Oslo algorithm, supports various capabilities including the computation associated with Boolean operations, such as the intersection of two arbitrary surfaces. B-splines are an ideal design tool, they are simple, yet powerful. It is also the case that many
common shapes can be represented exactly using rational B-splines. For example, all of the common primitive shapes used in CSG systems fall into this category. Other advantages include good computational and representational properties of the spline approximation: the variation diminishing property, the convex hull property and the local interpolation property. There are techniques for matching a spline-represented boundary curve against raw data. Although the final result may be an approximation, it can be computed to any desired precision (which permits nonuniform sampling).

The object shown in Figure 7 was rendered at orientations of 0, 22.5 and 45 degrees. An example image is shown of several objects in a scene. The synthesized logical sensor object merely sends a message to

the segment program for Camera 1 (a Fairchild 3000 CCD camera), then sends a message to each of the features used, then sends a message to the distance function object with the appropriate weights. The system has been implemented in PCLS (the Portable Common Lisp System) using objects and methods. The feature calculations are performed by running C code called from within the instances of the feature objects.

As another application, consider the synthesis of 3-D object recognition strategies. Consider the object shown in Figure 8.
@begin(figure)
@Blankspace(3inches)
@center(@b[Figure 8]. Polyhedron Designed in Alpha_1)
@end(figure)
It was designed with the Alpha_1 modeling system. Using the approach outlined above, the strategy tree shown in Figure 9 was produced. The tree consists of a top level node (not shown) representing the object, and a set of subtrees (shown in the figure) whose roots compose a set of dihedral edges which cover all aspects of the object.
@begin(figure)
@Blankspace(7inches)
@center(@b[Figure 9]. Strategy Tree for Polyhedron Recognition)
@end(figure)
The only feature used here is dihedral angle (indicated by the two faces at each node). No constraints were used other than the object's geometry. Given a
range data scene (see Figure 10), the edges were detected and the appropriate subtree which would be traversed during recognition is given in Figure 11. Representative views of the aspects are shown.
@begin(figure)
@Blankspace(3inches)
@center(@b[Figure 10]. Range Data of Polyhedron)
@end(figure)
@begin(figure)
@Blankspace(5inches)
@center(@b[Figure 11]. Strategy Tree Traversal)
@end(figure)
Clearly, non-polyhedral objects lead to a different kind of analysis, given that such objects tend to have more continuous features.

For an example of this approach applied to dextrous manipulation planning, see (Grupen 1987).
@Blankspace(1.5cm)
@u(Summary and Future Work)
@Blankspace(1cm)
The Multisensor Knowledge System offers many advantages for the design, construction, and simulation of multisensor systems. We have described many of those. In addition, we are currently working on a CAD-Based 3-D vision system. That is, we are developing a set of rules which will evaluate the 3-D geometry and function of any part designed with the Alpha_1 CAGD system. In this way, weak recognition methods can be avoided and specially tailored logical sensor objects can be synthesized automatically. Another area of current research interest is the simulation of multisensor systems. We believe that our approach can lead to very natural, straightforward, and useful simulations which can include native code running on the target processors. Finally, we are also investigating the organization of knowledge in the Multisensor Knowledge Base. Certain structuring of the data may lead to improved or simplified analysis.
@Newpage
@Blankspace(2)
@u(References)
@Blankspace(1)
@begin(format)
Albus, J. (1981) Brains, Behavior and Robotics. BYTE Books, Peterborough, New Ha

mpshire.
Bajcsy, R. (1984) GRASP:NEWS Quarterly Progress Report. Technical Report Vol. 2, No. 1, The
University of Pennsylvania, School of Engineering and Applied Science, 1st Quarter.
Booch, Grady. (1983) Software Engineering with Ada. Benjamin/Cummings Publishing Co., Menlo
Park, California.
Chiu, S.L., D.J. Morley and J.F.Martin. (1986) Sensor Data Fusion on a Parallel Processor. In Pro-
ceedings of the IEEE Conf. on Robotics and Automation, pp. 1629-1633. San Francisco, CA, April.
Fujimoto, R.M. (1985) The SIMON Simulation and Development System. In Proceedings of the 1985
Summer Computer Simulation Conference, pp. 123-128. July.
Grupen, Roderic A. and Thomas C. Henderson (1987) High-Level Planning for Dextrous Manipula-
tion. Tech. Report UU-CS-87-010, University of Utah, Dept. of Computer Science, April.
Henderson, T., E. Triendl and R. Winter (1980) Model-Guided Geometric Registration. Tech. Report
NE-NT-D-50-80, Deutsche Forschungs- und Versuchsanstalt f@act[udieresis]r Luft- und Raumfahrt.
Henderson, T.C. and E. Shilcrat. (1984a) Logical Sensor Systems. J. of Robotic Systems
1(2):169-193.
Henderson, T.C., E. Shilcrat and C.D. Hansen (1984) A Fault Tolerant Sensor Scheme. Proceed-
ings of the International Conference on Pattern Recognition, August, pp. 663-665.
Henderson, T.C., C.D. Hansen, and Bir Bhanu (1985a) The Specification of Distributed Sensing and
Control. Journal of Robotic Systems 2(4):387-396.
Henderson, T.C., Chuck Hansen and Bir Bhanu (1985b) A Framework for Distributed Sensing and
Control. In Proceedings of IJCAI 1985, pp. 1106-1109.
Henderson, T.C. and Steve Jacobsen (1986a) The UTAH/MIT Dextrous Hand. In Proceedings of
the ADPA Conf. on Intelligent Control Systems. Ft. Belvoir, Va., March.
Henderson, T.C., Chuck Hansen, Ashok Samal, C.C. Ho and Bir Bhanu (1986b) CAGD Based 3-D
Visual Recognition. In Proceedings of the International Conference on Pattern Recognition
pp. 230-232. Paris, France, October.
Henderson, T.C. (1987) Report on the NSF Workshop on Multisensor Integration in Manufactur-
ing Automation, Univ. of Utah Tech. Report, UUCS-87-006, March.
Mitiche, A. and J.K. Aggarwal (1986) An Overview of Multisensor Systems. SPIE Optical Com-
puting 2:96-98.
Organick, E.I., M. Maloney, D. Klass and G. Lindstrom (1983) Transparent Interface between
Software and hardware Versions of Ada Compilation Units. Technical Report UTEC-83-030,
University of Utah, Salt Lake City, Utah.
Overton, K. (1986) Range Vision, Force, and Tactile Sensory Integration: Issues and an Approach
In Proceedings of the IEEE Conference on Robotics and Automation, p. 1463. San Francisco,
California, April.
Shilcrat, E., P. Panangaden and T.C. Henderson (1984a) Implementing Multi-sensor Systems in a
Functional Language. Tech. Report UUCS-84-001, The University of Utah, Februar

y.
Shilcrat, E. (1984b) Logical Sensor Systems. Master's Thesis, University of Utah.
Swope, S.M. and R.M. Fujimoto. (1986) SIMON II Kernel Reference Manual. Technica
l Report
UUCS-86-001, University of Utah, May.
@end(format)