```
@Make(Article)
@Device(ln01)
@begin(comment)
@Use[Bibliography="/n/sp/s/u/tch/proposals/general.bib"]
@end(comment)
@Style(Spacing 1, LineWidth 4.625inches)
@begin(format)

@MajorHeading(Multisensor Integration in a Multiprocessor
Environment@foot(This work was
supported in part by NSF Grants MCS-8221750, DCR-8506393, and DMC-8502115.
))



@begin(center)
Thomas C. Henderson and Eliot Weitz
Department of Computer Science
The University of Utah
Salt Lake City, Utah 84112


@b[Abstract]

@end(center)
@end(format)
```

Multisensor systems offer much opportunity for concurrent operation.
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.
We give an example application of the system to CAD-based 2-D vision.

```
@Newpage
@Section(Introduction)
```

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. This topic has become the subject of more intense study, including a recent NSF-sponsored workshop on multisensor integration in manufacturing automation@cite[Henderson87b].

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(FullPageFigure)
@Blankspace(8inches)
@begin(format)

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

@end(format)
@end(FullPageFigure)
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@cite[Henderson84c],

functional (or applicative) style programming@cite[Shilcrat84],

features and their propagation through a network@cite[Shilcrat84a],

the specification of distributed sensing and
control@cite[Henderson85a,Henderson85h],

the automatic synthesis of Logical Sensor Specifications for CAD/CAM applications@cite[Henderson86c,Henderson86d].
@end(itemize)
Related work includes that of Albus@cite[Albus81] on hierarchical control, Bajcsy et al.@cite[Bajcsy84a] on the Graphical Image Processing Language, Overton@cite[Overton86] on
schemas, and Chiu@cite[Chiu86] on functional language and multiprocessor implementations. For an overview of multisensor integration, see Mitiche and Aggarwal@cite[Mitiche86].

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.

@Section(Objects and Methods)

Several distinct programming styles have been developed over the last few years, including: applicative-style programming, control-based programming, logic programming, and object-based programming.

Applicative style programming exploits function application as its main operation and regulates quite strongly the use of side-effects@cite[Henderson80].

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@cite[Booch83, Organick83].

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.

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: parallel processing, real-time control, exception handling, and unique I/O control.

Sensors typically require such operations as: enabling/disabling, limit setting, status checking, and periodic logging of state.

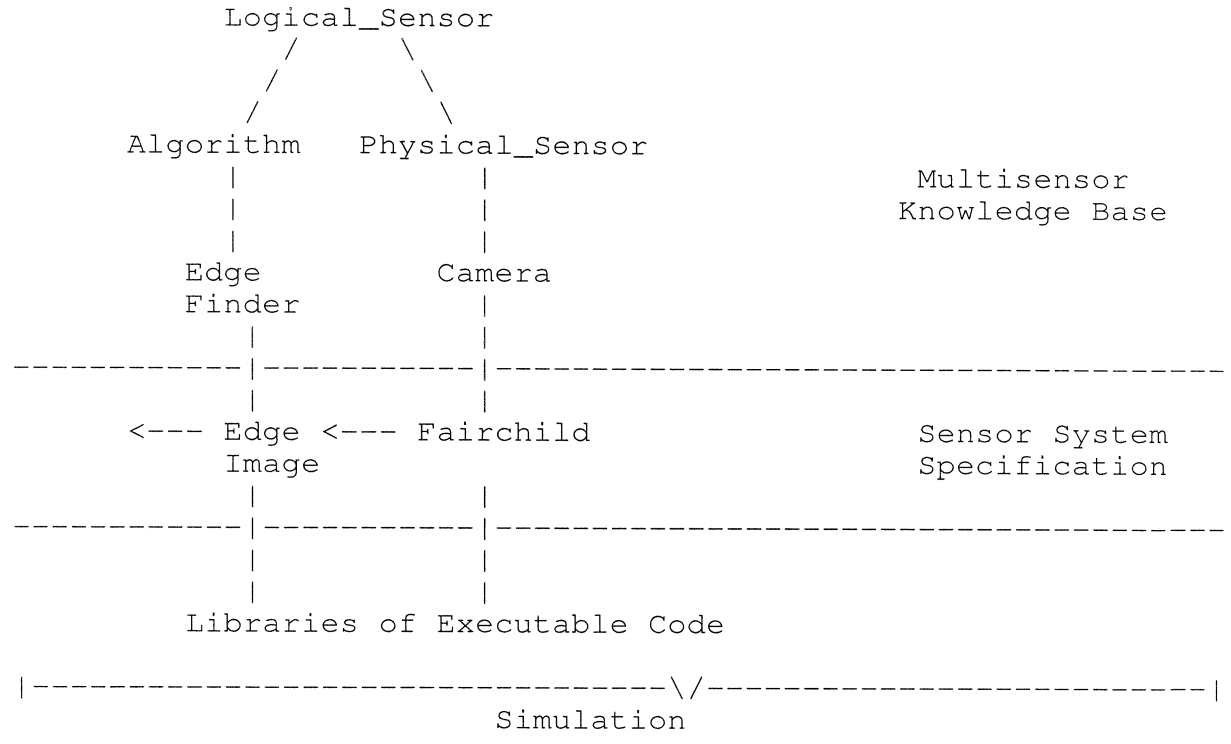
That is, sensor systems must respond to out-of-limit readings and issue alarms, detect faulty sensors, and recover from failure, and these functions can be implemented in a straightforward way.

@Section(Multisensor Knowledge Systems)
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(FullPageFigure)
@begin(comment)
@begin(verbatim)



@end(verbatim)
@end(comment)
@Blankspace(8inches)
@center(@b[Figure 2]. Multisensor Knowledge System)
@end(FullPageFigure)
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.

@SubSection(The Multisensor Knowledge Base)
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, a
@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, is 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.

In previous work, we have described a set of generally applicable physical sensor features@cite[Henderson84a]. 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.

@SubSection(Sensor Specification)
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.
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).

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.

@Section(An Example Application: CAD-Based 2-D Vision)

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 3 shows the scheme pictorially.
@begin(FullPageFigure)
@Blankspace(8inches)
@center(@b[Figure 3]. Synthesis of Part Detector)
@end(FullPageFigure)

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.

A 2D feature based vision system was created incorporating the idea of knowledge based (MKS) implemented in FROBS (FRames + OBjectsS)@cite(muehle86). Algorithms to perform segmentation and global feature calculation on binary images are written in C. The system uses the global feature method for object recognition with a weighted distance function for an n-dimensional

feature vector. The software exists on several
Vaxen and an HP 9000 series workstation supporting Common Lisp and FROBS.
Figure 4 shows the configuration of the inspection system.
The HP workstation relies on the Remote File Server (RFS) to link together
various parts of the knowledge base. The SP Vax is used as the principle
storage machine for the 2D vision algorithms written in C, the CS Vax due to
its greater calculation speed is used as the calculation component of the
system, and the GR Vax supports the Alpha-1 CAD system. Sensors in the
system include a Fairchild Automation Camera (CCD) and a Vicom image
processor. These sensors are controlled through ports on the SP Vax.

@begin(figure)
@blankspace(6 inches)
@center(@b[Figure 4.] 2D vision system overview)
@end(figure)

@subsection(FROBS)

FROBS (FRames + OBjectS) is a Lisp package which supports both frames and
objects @cite(muehle86). It is a "best of both
worlds" application, boasting the speed of Common Lisp objects and
the benefits of frame world. FROBS is written in HP Common Lisp and also
runs in PCLS. Frobs come in two basic flavors, class frobs and instance
frobs. Any operation which can be performed on one type of frob can be
performed on the other. The basic building block of the FROBS package
is called a module. Modules consist of a class frob and all of its
associated methods. This provides for total method and data access
hiding with no distinction between methods and slots. These modules
can be used as the building blocks for a 2D recognition system since they
can be mapped directly from a logical sensor specification of the system
or be the "machine code" output of a logical sensor "compiler." Class frobs
are created for each possible component in the system, algorithms, sensors,
processors, etc.

From this tree of class definitions instances can be created to represent
actual algorithms or pieces of hardware from which the inspection system is
synthesized. Class methods for "running" a frob instance can be invoked by
higher level instances as in the case of a segmenter frob invoking a camera
frob to obtain an image. Methods are written in Common Lisp and have full
access to the slots of a frob in their module. A package supporting forward
chaining rules and daemons for frobs is expected to be released shortly.
This will give a capability for rule based synthesis and operation of
multisensor systems with frobs.

@subsection(A Multisensor System Written in FROBS)

With FROBS a 2D knowledge based vision system was created. This system
has the ability to train on Alpha-1 generated objects to produce recognition
software. Since the system
uses global features for recognition, binary images from the CAD system are
sufficient for the training analysis. The object is rendered in several
orientations for training. This is done to select which features are most
robust given variations in position and orientation. The selected robust
features together form a model for that object. The features are given
weights proportionate to their robustness.

Once a model for an object is built, a group of frobs are created which will
be the recognition system for that object. The system is generated from the
model information by a set of Common Lisp functions. These functions create
instances of frobs representing the algorithms and sensors needed for
recognition of the model object. This includes the model object's robust
features algorithms as well as the necessary segmentation and recognition
software for each object recognizer. The
recognizer interface on the top level resembles a logical sensor which

accepts a common message via a frob method and produces an output vector as a result. In this case the output vector can be a list of distances for objects in a scene or a simple true/false for a single object given a threshold of acceptance. Commands are filtered down through the system hierarchy and output is filtered upward. Capabilities exist for dynamic restructuring of the system given a sensor or processor fault. Hardware is represented by hardware class frob instances which have methods to perform operational commands on them.

Methods which operate on any frobs within a module are primarily used for sensor operation and "frob to frob" communications. Most frobs have a @i(run) or @i(operate) method which executes some predefined command or lisp expression stored in an @i(executable) slot of the frob. The method has all of the other slot information available to it for access and modification. Most frob classes have an @i(output) slot to provide input to frobs outside of its module. An example of this is the @i(sensor) class frob which puts a pathname to a file containing the sensor's output. For the algorithm class frob, more specifically the feature subclass, a @i(run) method is used to invoke the algorithm on the defined processor with its input and output directed to and from defined locations. The algorithm class contains slots for @i(machine), @i(executable), and @i(output). An argument is passed to the method which is a frob containing input for the frob the method is being called on. A C-shell command in the @i(executable) slot is used along with @i(machine) and @i(output) slot information to form a single unix command which is then issued to the defined machine with input and output redirection specified.

The top level Logical Sensor class frob created by the recognizer synthesizer has a @i(program) slot containing a lambda expression for the sensor's operation. This expression mainly consists of method calls to instance frobs which represent the multisensor components of the recognition system. For example the object's model for the part is represented by an instance of a model class frob which holds such slot information as the object's robust features and the pathname for the model file which contains the actual model information. As previously mentioned the @i(run) methods operate on the frob instances using stdin/stdout file piping between executables on the target machines. This gives a multiprocessor capability since input, executable, and output files can exist on separate machines.

Several models were designed on Alpha-1 and recognizers for those models constructed by the frobs system. For most objects the most robust feature was the pixel count or area of the object in the rendered image. Area was weighted at over 90% for the distance function with the other 10% depending on invariant moments and a ratio between the object's axes of inertia. The system was tested by putting several milled pieces from the Alpha-1 system along with a milled piece of the model object on a back-lit table providing a silhouette image for the Fairchild camera. The camera's output was thresholded by the Vicom Image Processor and then processed by the part recognition software. By using mostly area for the recognition process the system was mostly successful in its selection of the correct object in the scene. Only objects of similar size presented a problem for the recognizer.

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.

@Section(Summary and Future Work)

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.