Logical Sensor Systems*

Tom Henderson and Esther Shilcrat
Department of Computer Science, The University of Utah, Salt Lake City, Utah 84112
Received March 16, 1984; accepted March 23, 1984

Multisensor 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 defined abstractly in terms of computational processes operating on the output from other sensors. Various properties of such an organization are investigated, and a particular implementation is described.

複数のセンサーを備えたシステムでは、それぞれのセンサーからの情報を、整然としかも効率的に取り扱えることが必要である。ここに提案する“論理的センサー装置”では、センサーからの出力を計算処理操作の一部とみなして概念的に取り扱うことができる。センサー系などの各種特性について検討し、特定の応用例について述べる。

I. INTRODUCTION

We describe and motivate a particular sensor system methodology, that of Logical Sensors, and its linguistic implementation, the Logical Sensor Specification Language. 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.[1–3]

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. For example, most robotic sensor-based systems to date have been designed around a single sensor or a small number of sensors, and *ad hoc* techniques have been used to integrate them into the complete system and for operating on their data. In the future, however, such systems must operate in a reconfigurable multisensor 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

*This work was supported in part by the System Development Foundation and NSF Grants ECS-8307483 and MCS-82-21750.

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, at least two issues regarding the configuration of sensor systems arise:

(1) how to develop a coherent and efficient treatment of the information provided by many sensors, particularly when the sensors are of various kinds;
(2) 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.

The Multisensor Kernel System (MKS) has been proposed as an efficient and uniform mechanism for dealing with data taken from several diverse sensors.[4–6] 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.[6] 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 are 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 multisensor systems provides a major motivation for the development of logical 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.[7] Our work has been done in the context of a robotic work station where the kinds of sensors involved include:

- 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.

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[8] and the Utah/MIT Dextrous Hand.[9] ATR systems integrate data from three (or more) sensors: microwave, FLIR, and LADAR. The Utah/MIT Hand includes a tactile sensing system which is composed of tactile element sensors gathered into tactile pads and placed on the Hand.

Other principal motivations for logical sensor specification are:

Benefits of data abstraction: the specification of a sensor is separated from its implementation. The multisensor 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[10] but is also an essential component of the system's reconfigurable nature. The inherent hierarchical structuring of logical sensors further aids system development.

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.

II. 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 so widely recognized in the area of graphics that 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 interacive graphics applications could be insulated from the peculiarities of the input devices of particular terminals, and thereby become portable."[11] 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 *type* of its output. Input devices so restricted are called *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 *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.[12] 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[13] 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.

III. LOGICAL SENSORS

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

A *logical sensor* is defined in terms of four parts:

(1) A *logical sensor name*. This is used to uniquely identify the logical sensor.

(2) A *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.

(3) A *selector* whose inputs are alternate subnets and an acceptance test name. 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.

(4) *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. *Each* alternate subnet in the list is itself composed of (a) a set of *input sources*. Each element of the set must either be itself a logical sensor, or the empty set (null). Allowing null input permits *physical* sensors, which have only an associated program (the device driver), to be described as a logical sensor,

thereby permitting uniformity of sensor treatment. (b) A *computation unit* over the input sources. Currently such computation units are software programs, but in the future, hardware units may also be used. In some cases, a special "do-nothing" computation unit may be used. We refer to this unit as PASS.

A logical sensor can be viewed as a network composed of subnetworks which are themselves logical sensors. Communication within a network is controlled via the flow of data from one subnetwork to another. Hence, such networks are *data flow* networks.

Alternatively, we present the following inductive definition of a logical sensor:

A logical sensor is an acceptance test which checks (sequentially and on demand) the output of either (base case 1):

(1) A list of computation units, with specified output type (the characteristic output vector), which require no input sources.
(2) A list of computation units, with specified output type, whose input sources are logical sensors.

Figure 1 gives a pictorial presentation of this notion. The characteristic output vector declared for this logical sensor is (x–loc:real, y–loc:real, z-loc:real, curvature:integer). We present two examples to clarify the definition of logical sensors, and in particular to show how the inputs to a logical sensor are defined in terms of other logical sensors and how the program accepts input from the source logical sensors, performs some computation on them, and returns as output a set (stream) of vectors of the type defined by the characteristic output vector. Figure 2 shows the logical sensor specification for a "camera" which happens to have no other logical sensor inputs. The specification for a stereo camera range finder called "Range_Finder" is given in Figure 3. The program "stereo" takes the output of the two cameras and computes vectors of the form (x, y, z) for every point on the surface of an object in the field of view. 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

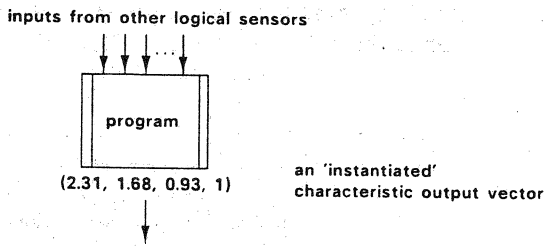
inputs from other logical sensors
program
(2.31, 1.68, 0.93, 1)
an 'instantiated' characteristic output vector

Figure 1. Graphical view of a logical sensor.

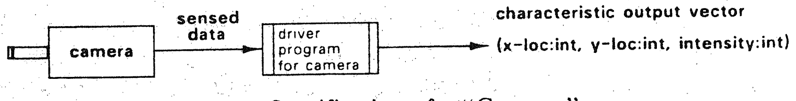
camera
sensed data
driver program for camera
characteristic output vector
(x-loc:int, y-loc:int, intensity:int)

Figure 2. The Logical Sensor Specification of a "Camera."

packaged as indicated. This allows the user to create "packages" of methods which produce equivalent data, while ignoring the internal configurations of those "packages."

A. Formal Aspects

Having described how logical sensors are developed and operate, we now define a logical sensor to be a *network* composed of one or more subnetworks, where each subnetwork is a logical sensor. The computation units of the logical sensors are the nodes of the network. Currently, the network forms a rooted directed acyclic graph. The graph is rooted because, taken in its entirety, it forms a complete description of a single logical sensor (versus, for example, being a description of two logical sensors which share subnetworks). We also say that it is rooted because there exists a path between each subnetwork 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 subnetwork 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 subnetworks. Hence, the networks described by the logical sensor specification language are data flow networks, and have the following properties:[14]

- A network is composed of independently, and possibly concurrently, operating subnetworks.
- A network, or some of its subnetworks, may communicate with its environment via possibly infinite input or output streams.
- Subnetworks are modular.

Since the actual output produced by a subnetwork may depend on things like hardware failures (and because the output produced by the different subnets of a logical sensor are only required to have the same type), the subnetworks (and hence the network) are also indeterminate.

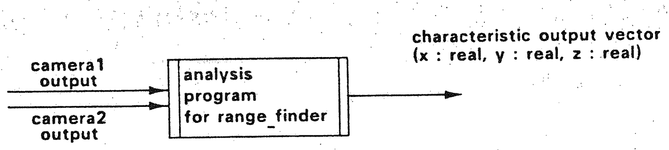
camera1 output
camera2 output
analysis program for range_finder
characteristic output vector
(x : real, y : real, z : real)

Figure 3. The Logical Sensor Specification of "Range_Finder."

B. Logical Sensor Specification Language

We have shown that a logical sensor has the following properties:

- A logical sensor is a network composed of subnetworks which are themselves logical sensors.
- A logical sensor may be defined only in terms of other, previously defined, logical sensors.
- A computation unit is an integral part of the definition of a logical sensor.
- A logical sensor produces output of the type declared by its characteristic output vector, and the declaration of the characteristic output vector is also an integral part of the definition of a logical sensor.

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.

1. Syntax

(logical-sensor)	→	(logical-sensor-name) (characteristic-output-vector) (selector) (alternate-subnet-list)
(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)}*
(acceptance-test-name)	→	(identifier)
(input-list)	→	(logical-sensor-list) | null
(logical-sensor-list)	→	(logical-sensor) {(logical-sensor-list)}*
(computation-unit-name)	→	(identifier)

2. Semantics

Below we present a high-level description of the *operational* semantics (i.e., the execution effect) for each rule of the grammar:

(1) A *logical sensor* declaration provides an associated name for the logical sensor used for identification purposes, a characteristic output vector to declare the type of output for that logical sensor. A selector performs the test and switch after the acceptance test and the alternate subnet list establishes the alternative ways of providing the characteristic output vector.

(2) A *logical sensor name* declaration associates a (unique) identifier for the logical sensor.

(3) A *characteristic output vector* declaration establishes the type of output for the logical sensor.

(4) A *name-type list* declaration establishes the precise nature of the output type as declared by the characteristic output vector. It consists of a cross product of types, with an associated name.

(5) A *selector* declaration specifies the order in which the alternates in the alternate subnet list will be tested by the acceptance test.

(6) An *alternate subnet list* declaration establishes a series of input sources, computation unit name tuples, thus making known which logical sensors and computation units are part of the definition of the logical sensor being declared.

(7) An *input list* declaration establishes which legal input sources (either none or a series of logical sensors) are to be used as input to the computation unit.

(8) A *logical sensor list* declaration establishes the set of logical sensors to be used as input.

(9) A *computation unit name* declaration establishes the name of the actual program which will execute on the declared input sources.

(10) A *acceptance test name* declaration establishes the name of the actual program which will be used to test the alternate subnets.

We are also currently working on providing more formal semantics for the logical sensor specification language. Many works provide *denotational* semantics (i.e., semantic schemes which associate with each construct in the language an abstract mathematical object) for general data flow networks.[14-16] When such semantics have been given for the networks represented by logical sensors, we will be able formally to prove desired network properties, such as that a network can execute forever.[15] We will also be able to prove that the output of a specified logical sensor has particular properties of interest (e.g., that its type matches that of the characteristic output vector).

C. 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[2] 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 Fig. 4). The operations allowed on logical sensors include:

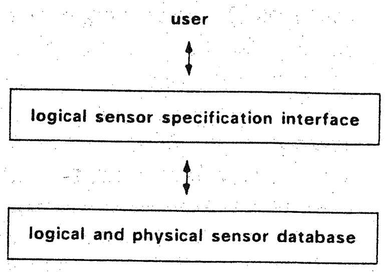
user
logical sensor specification interface
logical and physical sensor database

Figure 4. The Logical Sensor System Interface.

Create: a new logical sensor can be specified by giving all the necessary information and it is inserted in the database.

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.

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

Display: show all parts of a logical sensor or list all logical sensor names.

Dependencies: show all logical sensor dependencies.

Appendix A gives a sample session with the logical sensor specification interface.

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 5.

We have written a translator which converts the s-expressions in the database into abstract syntax trees for a Function Equation Language (FEL).[17] 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 data flow strategies.

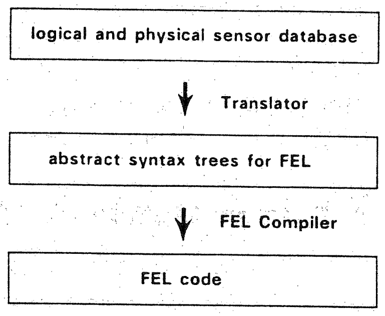
logical and physical sensor database
Translator
abstract syntax trees for FEL
FEL Compiler
FEL code

Figure 5. Steps to obtain executable code.

More on these topics can be found elsewhere.[3] 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. Moreover, the use of reduction allows us to store that part of the alternate subnet list which is currently backup in a skeletal form. Thus, a large savings in runtime space requirements may be achieved.

IV. 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 *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 *even deciding whether it arises from the hardware or software*" (emphasis added).[18] 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 *replication* or *replacement. Replication* means that exact duplicates of the failed component have been specified as alternate subnets. In *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."[18] 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.

A. Recovery Blocks

The recovery block is a means of implementing software fault tolerance.[18] 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 *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 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.[3] 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, two-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 6. 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.

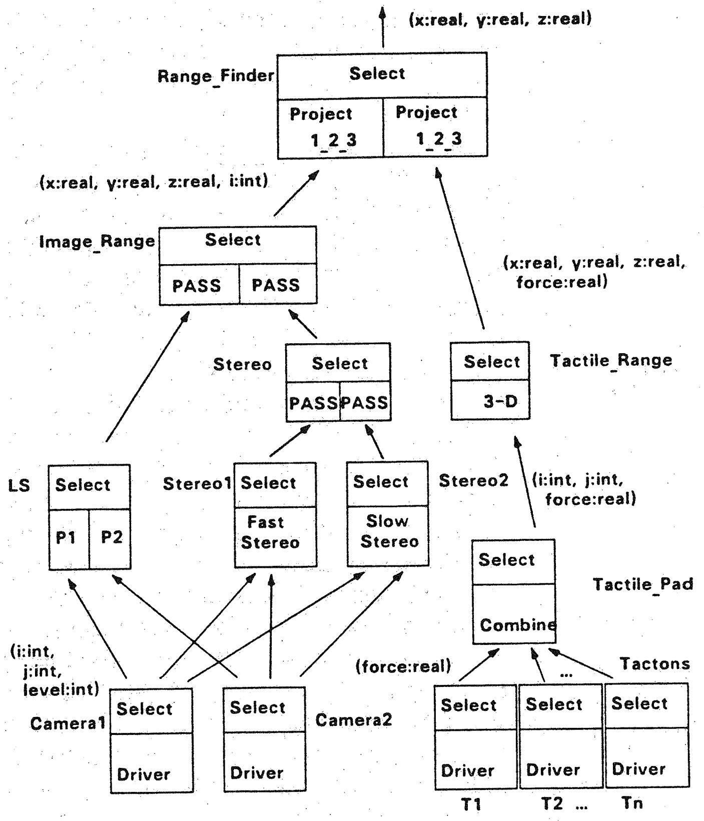

Figure 6. Logical Sensor Network for Range_Finder.

B. 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 *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 ±0.01%, and camera B has accuracy of ±0.04%. If the user has determined that the allowable limit on accuracy is ±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 ±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.

V. 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.

A. 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.[19] We have selected a set of features defined by Wright which we feel are generally applicable to physical sensors.

- *Error*—the difference between the value of a variable indicated by the instrument and the true value at the input.

- *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.
- *Repeatability*—the closeness of agreement within a group of measurements at the same input conditions.
- *Drift*—the change in output that may occur despite constant input conditions.
- *Resolution*—the smallest change in input that will result in a significant change in transducer output.
- *Hysteresis*—a measure of the effect of history on the transducer.
- *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.
- *Range*—the maximum range of input variable over which the transducer can operate.

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.

B. 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[8] 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 composition 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 of algorithms may be described. This problem is manifest in Bhanu's survey of the evaluation of automatic target recognition (ATR) algorithms.[8]

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., and more of that kind of work is required if we are to achieve comprehensive sensor systems.

VI. FUTURE RESEARCH: AUTOMATIC LOGICAL SENSOR GENERATION

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

A. Tupling/Merging Data

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 toward 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 defined automatically. There are two techniques under investigation, tupling and merging data.

1. Tupling Data

Tupling data is a technique which can be used to generate automatically 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 *bolt-detector* could be created by tupling the output of the logical sensors *edge-detector* and *hole-detector*. It should be noted that we assume that the latter two logical sensors produce output of the form (object No., feature 1, feature 2, . . . , feature *N*). For the sake of simplicity, in this example we assume that logical sensor *edge-detector* produces output of the form (object No., number of edges) and logical sensor *hole-detector* produces output of the form (object No., number of holes). Logical sensor *bolt-detector* will match an object number, and produce tuples of the form (object No., number of edges, number of holes).

2. Merging Data

Another facility we are investigating dynamically incorporates, in response to a system demand, a newly defined logical sensor which outputs the merge of three-dimensional logical sensor inputs. The idea is to accommodate an interactive request to allow the output of two physical sensors to be treated as one, for example, to create a multiple-view laser range-finder logical sensor from what had been two different laser range-finder logical sensors. In this example, a logical sensor *multiview-laser* is created with input logical sensors of both laser range finders, and the inputs are merged to produce output. Thus, the user can decide, interactively, to get more views without having to reconfigure the entire system. Also, such a facility obviates the need for having multiple program units where the only difference is in the number of expected inputs.

B. 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 *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 tells 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 with 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 configuration of the parameters on an algorithm which would make the demanded performance possible.

C. 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 *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 database–algorithm database 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 toward 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 how to improve themselves, and intend to investigate the use of training databases, and possible training database–algorithm database interaction schemes.

VII. CONCLUSION

We have defined a Logical Sensor Specification Language as a framework facilitating efficient and coherent treatment of information provided in multisensor systems. In addition to the issues raised when considering the language implementation itself, various extensions have been suggested. In particular, we have implemented:

(1) a Logical Sensor Specification Language compiler,
(2) general fault-tolerance features such as
  (a) a mechanism for detecting two types of sensor failure,
  (b) a technique by which switching to an alternate subnet is accomplished,
  (c) a method for determining when a sensor failure dictates top-level sensor failure,
(3) a database of physical sensors,
(4) automatic generation of tupling/merging logical sensors.

In addition, we intend to investigate 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; and training databases, and training database–algorithm database interaction schemes.

APPENDIX

The following session demonstrates the logical sensor specification system. Comments have been added in bold.

```
[PHOTO: Recording initiated Thu 8-Mar-84 1:19PM]

@psl
Extended 20-PSL 3.1, 15-Jun-83
1 lisp> (dskin "start.sl")

          Welcome to LSS!

Allowed options:

  1. Create Logical Sensor
  2. Update Logical Sensor
  3. Delete Logical Sensor
  4. Show Logical Sensor
  5. Show Dependency Graph
  6. Exit LSS
```

First, display all the existing logical sensors. They are either physical sensors or previously defined logical sensors.

```
Enter option number: 4
```
Show Logical Sensor

```
Enter name of logical sensor to be shown, or L for list: 1

  Defined logical sensors: CAMERA1 CAMERA2

Enter name of logical sensor to be shown, or L for list: camera1
  Logical sensor CAMERA1:
            Acceptor:  TIMEOUT
                COV:   (I:INT J:INT LEVEL:INT)
  Alternate Subnets: ((DRIVERCAM1 NIL))
```

The two logical sensors CAMERA1 and CAMERA2 are already known to the system. They are both physical sensors since they are leaf nodes.

```
Enter option number: 4
```

Show Logical Sensor

```
Enter name of logical sensor to be shown, or L for list: camera2

  Logical sensor CAMERA2 :
            Acceptor:  TIMEOUT
                COV:   (I:INT J:INT LEVEL:INT)
  Alternate Subnets: ((DRIVERCAM2 NIL))
```

Next, create the logical sensor laser-range-finder as shown in Figure 6 in the text

```
Enter option number: 1
```

Create Logical Sensor

```
Enter name of logical sensor to be created: laser-range-finder

Enter name of acceptance test: timeout

Enter characteristic output vector: (x:real y:real z:real intensity:int)

Enter program name: P1

Enter the list of input logical sensors: (camera1)

Do you want to define another subnet? (yes or no): y

Enter program name: P2

Enter the list of input logical sensors: (camera2)

Do you want to define another subnet? (yes or no): n
```

```
Enter option number: 4

Enter name of logical sensor to be shown, or L for list: laser-range-finder
  Logical sensor LASER-RANGE-FINDER:
                 Acceptor:  TIMEOUT
                      COV:  (X:REAL Y:REAL Z:REAL INTENSITY:INT)
  Alternate Subnets: ((P1 (CAMERA1)) (P2 (CAMERA2)))
```

Next, create the logical sensor stereo of Figure 6.

```
Enter option number: 1                        Create Logical Sensor

Enter name of logical sensor to be created: stereo

Enter name of acceptance test: timeout

Enter characteristic output vector: (x:real y:real z:real intensity:int)

Enter program name: ^V                        Hit wrong key. Fix later.

Enter the list of input logical sensors: (Camera1)

Do you want to define another subnet? (yes or no): y

Enter program name: slow-stereo

Enter the list of input logical sensors: (Camera1 Camera2)

Do you want to define another subnet? (yes or no): n
```

Display the logical sensors defined to this point.

```
Enter option number: 4                        Show Logical Sensor

Enter name of logical sensor to be shown, or L for list: l

  Defined logical sensors: CAMERA1 CAMERA2 LASER-RANGE-
  FINDER STEREO

Enter name of logical sensor to be shown, or L for list: stereo

  Logical sensor STEREO :
                 Acceptor:  TIMEOUT
                      COV:  (X:REAL Y:REAL Z:REAL INTENSITY:INT)
  Alternate Subnets: ((^V (CAMERA1)) (SLOW-STEREO (CAMERA1
  CAMERA2)))
```

Note the ^V for program name.

Correct the typo made when creating the logical sensor stereo. Also, correct the input list for the first alternative.

```
Enter option number: 2                    Update Logical Sensor

Enter name of logical sensor to be updated, or L for list: stereo

Add, delete or modify an alternate? (a, d or m): m

  Alternates defined for logical sensor STEREO are:
  Number   Alternate
  1        (^V (CAMERA1))
  2        (SLOW-STEREO (CAMERA1 CAMERA2))

Enter the NUMBER of the alternate you wish to modify: 1

Enter p to modify program name, i to modify input list: p

  Here is the alternate you wish to change: (^V (CAMERA1))
  Enter the new program name.

Enter program name: fast-stereo                Correct the program name.

More changes to this sensor? (y or n): y

Add, delete or modify an alternate? (a, d or m): m

  Alternates defined for logical sensor STEREO are:
  Number   Alternate
  1        (FAST-STEREO (CAMERA1))    Stereo requires 2 cameras.
  2        (SLOW-STEREO (CAMERA1 CAMERA2))

Enter the NUMBER of the alternate you wish to modify: 1

Enter p to modify program name, i to modify input list: i

  Here is the alternate you wish to change: (FAST-STEREO (CAMERA1))
  Enter the new input list.

Enter the list of input logical sensors: (camera1 camera2)

More changes to this sensor? (y or n): n
```

Display the updated logical sensor.

```
Enter option number: 4                    Show Logical Sensor

Enter name of logical sensor to be shown, or L for list: stereo

  Logical sensor STEREO :
           Acceptor:  TIMEOUT
               COV:   (X:REAL Y:REAL Z:REAL INTENSITY:INT)
  Alternate Subnets: ((FAST-STEREO (CAMERA1 CAMERA2)) (SLOW-
  STEREO (CAMERA1 CAMERA2)))
```

Create the logical sensor image-range of Figure 4.

```
Enter option number: 1                    Create Logical Sensor

Enter name of logical sensor to be created: image-range

Enter name of acceptance test: timeout

Enter characteristic output vector: (x:real y:real z:real intensity:int)

Enter program name: pass

Enter the list of input logical sensors: (stereo)

Do you want to define another subnet? (yes or no): n
```

Display the logical sensor image-range.

```
Enter option number: 4                    Show Logical Sensor

Enter name of logical sensor to be shown, or L for list: image-range

  Logical sensor IMAGE-RANGE :
           Acceptor:  TIMEOUT
               COV:   (X:REAL Y:REAL Z:REAL INTENSITY:INT)
  Alternate Subnets: ((PASS (STEREO)))
```

Next, we add an alternative subnet to an existing sensor.

```
Enter option number: 2                    Update Logical Sensor

Enter name of logical sensor to be updated, or L for list: image-range

Add, delete or modify an alternate? (a, d or m): a
```

```
  Alternates defined for logical sensor IMAGE-RANGE are:
  Number  Alternate
  1       (PASS (STEREO))

Enter the NUMBER you wish for the new alternate: 1
```

The switching order can be rearranged.

```
Enter progam name: pass

Enter the list of input logical sensors: laser-range-finder

  Inputs must be a list. What you gave LASER-RANGE-FINDER is not

Enter the list of input logical sensors: (laser-range-finder)

More changes to this sensor? (y or n)n
```

Display the new version of the logical sensor.

```
Enter option number: 4
```

Display Logical Sensor

```
Enter name of logical sensor to be shown, or L for list: l

  Defined logical sensors: CAMERA1 CAMERA2 LASER-RANGE-
                           FINDER STEREO IMAGE-RANGE

Enter name of logical sensor to be shown, or L for list: image-range

  Logical sensor IMAGE-RANGE:
            Acceptor:  TIMEOUT
                 COV:  (X:REAL Y:REAL Z:REAL INTENSITY:INT)
  Alternate Subnets: ((PASS (LASER-RANGE-FINDER)) (PASS (STEREO))

Enter option number: 5

  Dependency Table:

  CAMERA1                   0   0   1   1   1
  CAMERA2                   0   0   1   1   1
  LASER-RANGE-FINDER        0   0   0   0   1
  STEREO                    0   0   0   0   1
  IMAGE-RANGE               0   0   0   0   0
```

Exit the system. The sensor database is automatically updated.

```
Enter option number: 6              Exit LSS

Your requests have been handled!

NIL
NIL
2 lisp> (quit)
@pop

[PHOTO: Recording terminated Thu 8-Mar-84 1:29PM]
```

References

1. C. Hansen, T. C. Henderson, E. Shilcrat, and Wu So Fai, "Logical Sensor Specification," in *Proceedings of SPIE Conference on Intelligent Robots,* SPIE, New York, November 1983, pp. 578–583.
2. T. C. Henderson, E. Shilcrat, and C. Hansen, *A Fault Tolerant Sensor Scheme,* Computer Science UUCS 83-003, University of Utah, November 1983.
3. E. Shilcrat, P. Panangaden, and T. C. Henderson, *Implementing Multi-sensor Systems in a Functional Language,* Tech. Rep. UUCS-84-001, The University of Utah, February 1984.
4. T. C. Henderson and Wu So Fai, "A Multi-sensor Integration and Data Acquisition System," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* IEEE, New York, June 1983.
5. T. C. Henderson and Wu So Fai, *Pattern Recognition in a Multi-sensor Environment,* UUCS-83-001, University of Utah, July 1983.
6. Wu So Fai, "A Multi-sensor Integration and Data Acquisition System," Master's thesis, University of Utah, June 1983.
7. W. R. Nelson, "REACTOR: An Expert System for Diagnosis and Treatment of Nuclear Accidents," in *Proceedings AAAI-82,* August 1982, pp. 296–301.
8. Bir Bhanu, "Evaluation of Automatic Target Recognition Algorithms," in *Proceedings of the SPIE West '83,* SPIE, New York, August 1983.
9. S. Jacobsen, J. E. Wood, D. F. Knutti, and K. Biggers, "The Utah/MIT Dextrous Hand," in *MIT/SDF IRR Symp.,* August 1983.
10. N. Wirth, "On the Composition of Well-Structured Programs, in *Classics in Software Engineering,* E. N. Yourdan, Ed., Yourdon Press, London, 1979, pp. 153–172.
11. D. S. Rosenthal, J. C. Michener, G. Pfaff, R. Kessener, and M. Sabin, "The Detailed Semantics of Graphics Input Devices," *Comput. Graphics,* **16**(3), 33–38 (July 1982).
12. D. R. Olsen, and E. P. Dempsey, "SYNGRAPH: A Graphical User Interface Generator," in *SIGGRAPH '83 Conference Proceedings,* ACM, New York, July 1983, pp. 43–50.
13. M. D. Donner, "The Design of OWL: a language for walking," *ACM SIGPLAN Notice,* **18**(6), 158–165 (June 1983).
14. R. M. Keller, "Denotational Models for Parallel Programs with Indeterminate Operators," in *Formal Descriptions of Programming Concepts,* E. J. Neuhold, Ed., North Holland, Amsterdam, 1978, pp. 337–366.
15. G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proceedings of IFIP,* 1974, pp. 471–475.
16. G. Kahn, and D. MacQueen, "Coroutines and networks of parallel processes," in *Proc. IFIP 77,* 1977, pp. 993–998.

18. B. Randell, *System Structure for Software Fault Tolerance,* Prentice-Hall, Englewood Cliffs, NJ, 1977, pp. 195–219.
19. J. D. Wright, *Measurements, Transmission, and Signal Processing,* Van Nostrand Reinhold, New York, 1983, pp. 80–112.
20. G. Beni, S. Hackwood, L. A. Hornak, and J. L. Jackel, "Dynamic Sensing for Robots: An Analysis and Implementation," *Robot. Res.,* **2**(2), 51–60 (Summer 1983).