Implementing Multi-Sensor Systems in a Functional Language[1]

Esther Shilcrat, Prakash Panangaden and Tom Henderson

UUCS-84-001
24 February 1984

Department of Computer Science
The University of Utah
Salt Lake City, Utah 84112
(Shilcrat, Panangaden, Henderson)@UTAH-20

Abstract: We discuss a methodology for configuring systems of sensors using a functional language. To date no such general methodology exists, and existing multi-sensor systems have been hand-crafted around a particular application. Our main point is that the use of abstraction and of functional language features leads to a natural and simple approach to this problem. Our work exploits features of a particular functional programming environment, Function Equation Language (FEL) running on the REDIFLOW simulator, to develop a simple fault-tolerance scheme that avoids complicated issues of state restoration and switching protocols, and to develop implementations of multi-sensor systems that are very close to the abstract system specification and are hence manifestly correct.

keywords and phrases: Functional Programming, Sensor Systems, Fault Tolerance, Demand Driven Evaluation, Logical Sensors, Data Abstraction.

---

[1]This work was supported in part by NSF Grants ECS-8307483 and MCS-82-21750.

1. Introduction

Functional programming has long been advocated as a convenient and flexible method to facilitate the design and development of systems that are composed of independently defined modules functioning co-operatively. By and large, however, functional programming applications have been limited to situations that are well understood in an imperative setting. In this paper we examine a novel application of functional programming, namely a system for configuring networks of sensors. We use several features of functional languages to arrive at a simple and natural implementation of sensor networks. Indeed, our major thesis is that functional programming concepts are naturally suited to this important application area and that some common features of functional languages can be used to finesse certain problems that would arise in a more conventional programming environment. Furthermore, our fault-tolerance mechanism, though developed in our particular problem domain, has general applicability. Finally, our system is general in that it can be used with any multi-sensor system instead of being "hand-crafted" around a particular concrete application.

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. 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.

Other examples of sophisticated sensor systems include automatic target recognition (ATR) systems [Bhanu 83] and the Utah/MIT Dextrous Hand [Jacobsen 83]. ATR systems integrate data from three sensors: microwave, FLIR (Forward Looking Infra-Red), and LADAR (Low Altitude Detecting and Ranging), via seven separate computational steps. 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.

The increasing sophistication of sensor applications has raised a variety of problems which cannot be efficiently solved by the ad hoc configuration techniques generally used in the past for

single sensor systems. For instance, in multi-sensor systems it is much more difficult to correctly express the inherent relations of transformed data. In ATR systems, for example, the lowest level would be the raw data returned by the physical sensor, next, the data which results from preprocessing the raw data, then the result of the next software layer, target detection, and so on. Often there are relationships to be expressed within a level; for example, preprocessing may involve first transforming the data by a high pass filter program, and then by locally variable scaling. In the case of the Utah/MIT Hand, the most obvious relationships to be expressed are between the physical sensors themselves: the tactile elements form a pad, a group of pads form a finger (or palm), the fingers and palm form the Hand.

Thus, in multi-sensor systems, the complexity introduced by the use of many sensors and software layers makes paramount the need for a coherent and systematic treatment of data, particularly when using various kinds of sensors. Multi-sensor systems also complicate the issue of fault tolerance. In single sensor systems, backup sensors would generally be duplicates of the failed sensor, or would be functionally equivalent to it, and would be permanently mounted and running. In the case of multi-sensor systems, mounting duplicate sensors may be either prohibitively expensive or impossible due to physical space limitations. Furthermore, stopping a system in order to mount a replacement sensor will conflict with the real-time expectations of some sensor systems.

A solution to these problems can be achieved through the use of logical sensor specification, a framework in which sensors and their software layers can be abstractly defined in terms of computational processes operating on the output from other sensors. Such abstractions are called logical sensors. Logical sensors are a means of information hiding such that sensor system configuration becomes tantamount to function composition and application.

The language we use is Function Equation Language (FEL) [Keller 82] which runs on the REDIFLOW multiprocessor simulator [Keller 83]. The REDIFLOW simulator supports both **graph reduction** and **dataflow** evaluation strategies, with graph reduction being the default. The invocation of functions is controlled in a demand-driven fashion. This combination is ideally suited to our applications, especially as regards the fault-tolerance mechanism. This particular language embodies a functional semantics based on the concept of Function Graphs [Keller 77]. This graphical view of functions is ideally suited to describing systems which are networks of computing agents as is the case in our multi-sensor systems.

2. Logical Sensors

Sensor system configuration in multi-sensor systems presents a variety of problems, including how to express the relationships between sensed/transformed data, how to focus on the issues of importance amidst the confusing plethora of details concerning both the sensors and associated software, and how to express and implement fault tolerance. We present logical sensor specification, a methodology designed to deal with the particular problems inherent in multi-sensor system configuration. The need for this abstraction mechanism was originally articulated in the course of developing the Multi-sensor Kernel System (MKS) [Hansen 83, Henderson 83a, Henderson 83b, Wu 83]. Logical sensor specification incorporates solutions to the above mentioned key problems of multi-sensor systems, via information hiding, data relationship expression and alternate computation paths.

Many sensor/software details are irrelevant in determining sensor system configuration. However, one item of information which is always necessary is the type of output produced. Thus, logical sensors are designed as a means by which to insulate the designer from unnecessary details, while keeping necessary information visible. This is accomplished by creating "packages" of sensors, wherein only output type is visible to the rest of the system. Thus, a logical sensor, however complicated, is viewed simply as an object of a specified type.

This concentration on type not only serves to shield the designer from unnecessary information, but also facilitates the view of system design as function application. As a simplified example of this, consider a sensor system for finding edges designed around a single camera. The camera is a logical sensor, and hence object, of type (stream[2] of i:int, j:int, intensity:int). The overall logical sensor, "edge-finder", is an object of type (stream of i:int, j:int, edge:bool). Thus, a transducer of type (stream of int x int x int) ---> (stream of int x int x bool) is needed to create "edge-finder" out of "camera" (see figure 3-2).

The following inductive definition shows how logical sensors hide unnecessary information and express fault tolerance:

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

---

[2]Since physical sensors continually produce output, which is continually being processed, both physical and logical sensors produce streams of output.

2. A **characteristic output vector**. A vector of types which serves as a description of the output vectors that will be produced by the logical sensor.

3. A **selector** whose inputs are alternate subnets (below) and an provided acceptance test. 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. Each alternate subnet in the list is itself a pair composed of:

    * A tuple of **input sources**. Each input source is in turn itself a logical sensor. As a special case, the tuple of input sources may be 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.

    * A **computation unit**, a transducer, over the input sources. In some cases, a special "do-nothing" unit may be used. We refer to this unit as PASS.

Since even physical sensors are classified as logical sensors, a logical sensor can be viewed as a network composed of sub-networks which are themselves logical sensors. Communication within a network is controlled via the flow of data from one sub-network to another. The action of the computation units is to accept a stream of input data and produce a stream of output values; thus, a logical sensor may be viewed as a **data-flow** network.

3. Functional Fault Tolerance

Multi-sensor systems present a challenging opportunity to turn what is in one case a source of weakness (the number and variety of sensors) into a source of strength in terms of building fault tolerant sensor systems [Henderson 83c]. Through the use of data abstraction in a functional environment, logical sensor specification helps provide ways in which to meet this challenge.

By focusing concentration on the object type, logical sensors facilitate the recognition of ways in which to make sensors already in the system serve as backups in addition to their primary sensing role. As a simplified example, consider logical sensor "tactile_pad," of type (stream of i:int, j:int, force:bool), which contains three tactile element logical sensors of type (stream of force:bool). Suppose that one of the three tactile elements fail. If an interpolator function of type (stream of bool x bool) ---> (stream of bool) can be created or composed, the other two tactile elements could be used as backups for the failed element. As this example illustrates, backups may well not be simple replacement of sensors, but replacements which involve one or more sensors, and one

or more software modules. Such replacements make it yet more desirable to abstract sensor-software combinations, and to avoid unnecessary consideration of the internal details of an abstract sensor.

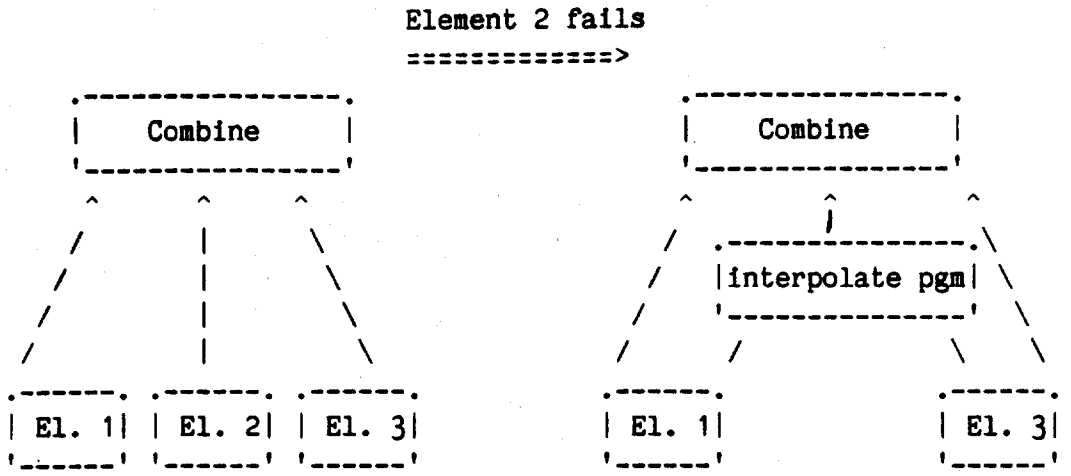


Figure 3-1: A Fault Tolerant Configuration for a Tactile Pad

We introduce the use of a **selector** node as a means to achieve fault tolerance in a functional environment. The selector determines which function application, i.e. alternate subnet, is acceptable, thereby allowing failure and recovery. The failures detected by a selector may be due to either hardware or software failures; no attempt to determine the exact cause of failure is made. This is in accordance with our view of the need for a hardware/software transparency in sensor systems, which is reinforced by the consideration that 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.

The selector node operates in a manner similar to that of the recovery block [Randell 77]. Like the recovery block, a selector contains a series of alternates which are to be tried in the order listed and uses an acceptance test to ensure that the output produced by an alternate is correct or acceptable. If the acceptance test passes the output from the first alternate, that stream of output data flows up, otherwise the second alternate is activated. If the output from the second alternate is not acceptable, then the third is tried, and so on. If no alternate passes, the selector reports failure. Switching may then be accomplished by some higher level selector.

However, selectors do not entail the the use of complicated error recovery mechanisms (restoring the state, and so on), which is typical in imperative settings. In an imperative program a sensor may update global variables before failing; such variables have to be reset before a backup can be

activated. In our case this need not be done since there are no global variables of any kind. When an alternate is activated by a selector it can start functioning immediately and the old network is eventually removed by the garbage collector.

This point is particularly important in our application, with regard to both time and space considerations. Not only may the time taken to restore the state conflict with the real-time nature of many sensor systems, but sensor systems also generally pose space problems by the large amount of data they produce; such a situation is only exacerbated by the need to store state recovery information. In addition, most sensor systems are designed to sense the current environment, in accordance with the view that the scene to be sensed is in a constant state of flux. Thus, it is useless to try to "roll back" to a previous time-slice; since the scene is changing, only the location of objects at the current moment is of interest. The complexity of using a global state in such sensing lends itself to dangerous inconsistencies, rendering the functional approach both more natural and safer.

We now present the formal definition of some of our example logical sensors:

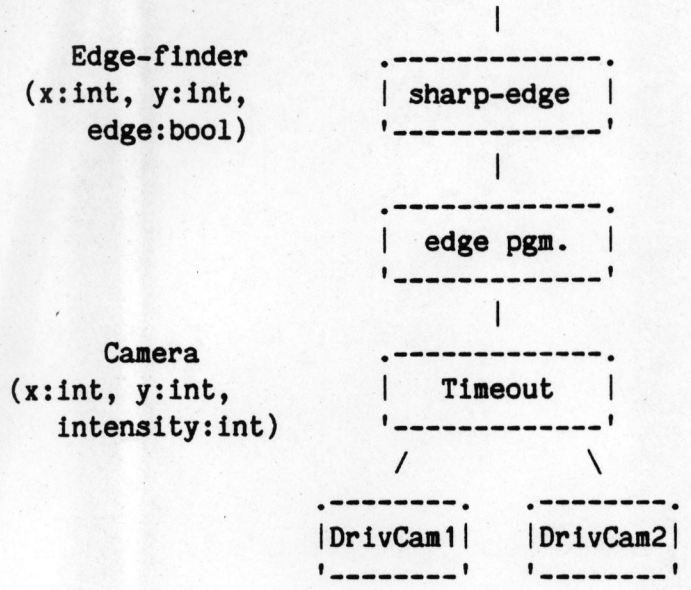
```
Logical Sensor Name: Edge-finder
Characteristic Output Vector: x:int, y:int, edge:bool
Acceptance Test: Sharp-edge
Subnet(s) (computation unit, input source pairs):
1. (Edge-program,Camera)
```

This logical sensor has only one subnet, and hence any failure must be accommodated by some higher level logical sensor. The input source, Camera, must itself be a logical sensor:

```
Logical Sensor Name: Camera
Characteristic Output Vector: x:int, y:int, intensity:int
Acceptance Test: Timeout
Subnet(s):
1. (Driver-camera1, null)
2. (Driver-camera2, null)
```

Camera is defined to have two alternate subnets, so that a failure of the physical camera1 will initiate a demand on physical camera2. Pictorially, logical sensor Edge-finder looks like:

Figure 3-2: Logical Sensor Edge-finder



4. Logical Sensor Systems in Function Equation Language

In this section we describe how the logical sensor system described in the last section is implemented in Function Equation Language (FEL) [Keller 82]. First we describe briefly some of the key features of the FEL/REDIFLOW environment that are of interest in our implementation.

One can view an FEL program in two semantically equivalent ways: (i) a textual form in which the programs are actually written; and (ii) a graphical form which is convenient for actual program design and composition. We shall discuss both of these views in what follows.

An FEL program is a function from a domain of inputs to a domain of outputs. Each such function is viewed as the node of a graph with the input arcs representing the data values on which the function is applied and the output arcs representing the data values resulting from the function application. The main graph defining a particular function can be supplemented by other graphs defining the auxiliary functions used in the top-level definition. Thus program composition is easily modelled as the linking together of graphs.

Syntactically an FEL program consists of a set of equations called an equation group. Each equation group also contains a result expression. Each equation in the equation group equates an identifier with an expression defining an object. These objects may be functions or any data structure, thus functions are first class objects in FEL. The effect of executing an FEL program is to

evaluate the result expression. Right-associative function application is denoted by an infix colon, ":", while left-associative function application, or currying, is denoted by a vertical bar, "|". Function abstraction is written with a "=>", thus λx.f(x) would be written as "x => (f:x)". Notationally an FEL equation group appears as:

{equations RESULT expression}.

The keyword RESULT indicates that the following expression is the result expression. The basic constructor functions used are "^" which is essentially an infix cons, "^ ^" which is essentially an infix append and "[,]" which is the tuple constructor. These are all lazy constructors and hence can be used with potentially infinite streams.

The standard evaluation mechanism for FEL is graph reduction [Trelevan 82]. The high-level textual FEL programs are mapped into a low-level functional language based on the graphical representation discussed above. The application of a function is modelled by macro-substitution of a node labelled with the function name by a graph which defines the meaning of the function. Reduction entails the creation of a new copy of a function body for each invocation of the function. The REDIFLOW simulator, however, also allows functions to behave as a stream-transducers (by adding the keyword PROCESS to the FEL function definition). Thus we may use both reduction and dataflow evaluation in the same network. We use reduction to ensure that those parts of the network that are present as backups are represented only in skeletal form until they are needed, and we use dataflow we ensure that a function acting on input streams does not get recopied for every member of the stream. Both evaluation mechanisms embody a demand driven strategy, that is, no expression is evaluated unless its result is needed for the evaluation of the outermost result expression.

To summarize, the main features of FEL/REDIFLOW that we exploit in the implementation of a logical sensor network are:

* the correspondence between the graphical view of logical sensors and the function graph view of FEL programs. This engenders a straightforward translation between specifications written in the Logical Sensor Specification Language (LSSL) and FEL code.

* the purely functional nature of FEL. This leads to a simple fault-tolerance scheme where complicated issues of state restoration are finessed.

* demand-driven function invocation. This means that nodes will be inactive until they are demanded by the selector and thus no unneeded data will be produced. Furthermore, switching between a logical sensor and its backups occurs entirely by the

propagation of demands, thus no explicit switching protocols are required.

* the hybrid evaluation strategy. This allows optimal use of resources since demanded nodes are not duplicated (dataflow) and nodes which are not demanded are not expanded (reduction).

Figure 3-2 makes manifest the following correspondences: A logical sensor is simply a function graph. A logical sensor name is the identifier which is equated to the FEL expression defining the graph in question. The characteristic output vector is simply the type of the function graph. The computation units and selectors correspond to functions, the interconnections between selectors and computation units are expressed as function applications. We now give a detailed example of the correspondence between logical sensors and FEL code. Consider the logical sensor shown in fig 3-2. The sensor picture presented translates directly to this FEL code:

```
Edge-finder = (selector|acc_sharp_edge):[Edge-program:Camera]
Camera = (selector|acc_timeout):[[Driver-camera1:[]],[Driver-camera2:[]]]

(* where the definition of selector is *)
selector = (acc_test => (subnet-list =>
           if subnet-list = [] then []
else
(acc_test:(first:subnet-list))^^((selector|acc_test):(rest:subnet-list))

(* the acc functions are constructed from the given (boolean valued)
acceptance tests.  We show acc_sharp_edge in the example below *)
PROCESS acc_sharp_edge:stream =
 if stream = [] then [] else
   if sharp_edge:(first:stream) then
    (first:stream)^(acc_sharp_edge:rest:stream)
    else []
```

Thus, the FEL equational definitions correspond exactly to the graphical view of logical sensor networks. In the graphical view, a logical sensor is a data stream which is formed by choosing (via the acceptance test) between computations preformed on other logical sensors. In the FEL view, a logical sensor is a stream of data (and hence an object) produced by applying the higher-level selector function to the acceptance test, and the list of computation units applied to logical sensors.

The selector function involves both dataflow and reduction style evaluation. The selector takes as arguments an acceptance test and a list consisting of computation units applied to their input

sources (the alternate subnet list). We assume that a correctly operating logical sensor produces an **infinite** stream of output. Thus an end of stream (represented by [] in FEL) signifies that none of the alternates produced acceptable data. This is known to be the case when the subnet list is empty. The acceptance test function is a filter on the incoming data stream constructed from the given (boolean valued) acceptance test; it passes on acceptable data and returns [] when a failure is detected. In order to operate as stream transducers they are implemented as processes. Reduction style evaluation enters through the use of "^ ^". As long as its first argument is producing data the second argument is not demanded since "^ ^" is lazy. If the first argument returns [] (indicating that that subnet has ceased to function) then the second argument of "^ ^" is demanded; until such time the subnets corresponding to the second argument of the "^ ^" are present only in skeletal form.

We now present a more detailed example. Range_Finder fig4-1 obtains surface point data via two 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 their first three elements. The preferred subnet contains logical sensor Image_Range. This logical sensor has two alternate subnets, both of which use the computational unit PASS. PASS does not affect the type of the logical sensor, being merely the identity function. These alternatives will be selected in turn to produce streams of type (x:real,y:real,z:real,i:integer). If both alternates fail Image_Range has failed. Range_Finder then selects the second subnet to obtain the (x:real,y:real,z:real) information from Tactile_Range. If Tactile_Range also fails, then Range_Finder fails.

The FEL code corresponding to this logical sensor network is shown below:

```
{ RESULT range_finder

range_finder =
(selector|acc_rf):[project123:image_range,project123:tactile_range]

image_range = (selector|acc_image):[PASS:laser_light,PASS:stereo]

tactile_range = (selector|acc_tr):[3_D:tactile_pad]

laser_light = (selector|acc_ll):[P1:camera_1,P2:camera_2]

camera_1 = (selector|acc_c1):[driver_camera_1:[]]

camera_2 = (selector|acc_c2):[driver_camera_2:[]]
```

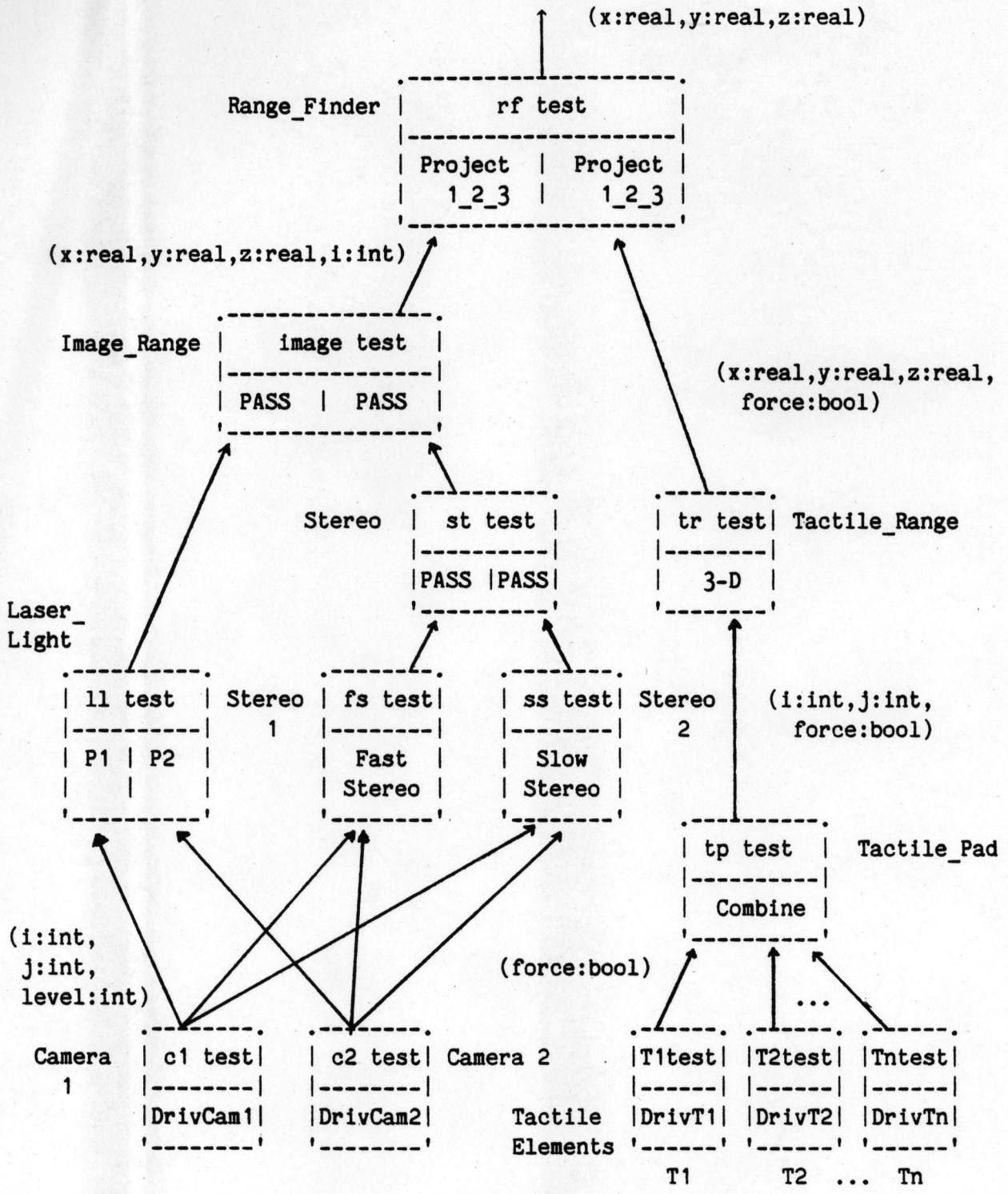


Figure 4-1: Logical sensor Range_Finder

```
stereo = (selector|acc_st):[Pass:fast_stereo,Pass:slow_stereo]

fast_stereo = (selector|acc_fs):[fast_stereo_prog:[camera_1,camera_2]]

slow_stereo = (selector|acc_ss):[slow_stereo_prog:[camera_1,camera_2]]

tactile_pad = (selector|acc_tp):[combine_prog:[t1,...,tn]]

t1 = (selector|acc_t1):[driver_t1:[]]}
```

Working with a graph reduction system allows us to reuse subnetworks via structure sharing. Thus for example the camera_1 and camera_2 logical sensor subnetworks are used by laser_light, fast_stereo and by slow_stereo. Since activation of a subnet is accomplished by the arrival of demand, we need not articulate any protocol for switching between networks. With several levels of selection, as in this example, it would be tedious to explicitly encode such switching. In the present example the selection might occur as follows: The top-level node (select_rf) would demand the reduction of image_range, if the data that appeared passed the acceptance test (acc_rf) then this would be reported as the result of running this logical sensor network, if it did not pass the acceptance test then the reduction of tactile_range would be demanded. When the reduction of image_range is demanded, the selector at the top of the corresponding function graph would demand the evaluation of laser_light. If laser_light failed to produce acceptable data, stereo would be demanded.

5. Conclusion

A prototype logical sensor specification system is currently under development. This features an interactive front-end which allows the designer to specify a logical sensor in an abstract specification language, which is then compiled into FEL. The front-end and the compiler are written in Portable Standard Lisp (PSL). Further system development awaits the completion of the REDIFLOW project.

Several theoretical and pragmatic issues remain to be developed. The use of acceptance tests introduces indeterminacy; this should be addressed by a formal semantics for logical sensors. Our formalism could be extended to allow recursive definitions of logical sensors, since this is required to describe sensor systems which involve feedback loops. For a more refined approach, one could expand the notion of type to include performance characteristics such as accuracy or resolution. These characteristics can be used to determine a more accurate set of functionally equivalent logical sensors. The feasibility of incorporating and propagating such information through a logical

sensor network is being investigated.

To summarize, we have developed a functionally based, general purpose, multi-sensor configuration system. To date, no such system has been developed in an imperative setting. We have argued that the functional programming style leads to a modular and semantically transparent multi-sensor system design. In particular, we have shown that, for our purposes, fault tolerance is expressed more naturally in a functional framework than in the standard imperative style recovery block schemes. We have shown how the use of a demand driven control protocol completely obviates the need for explicit articulation of switching protocols. Finally, by using the novel REDIFLOW architecture which allows both reduction and dataflow evaluation, we minimize overheads traditionally associated with functional languages.

References

[Bhanu 83] Bir Bhanu.
Evaluation of Automatic Target Recognition Algorithms.
In Proceedings of the SPIE West '83. August, 1983.

[Hansen 83] Hansen, C., T.C. Henderson, Esther Shilcrat and Wu So Fai.
Logical Sensor Specification.
In Proceedings of SPIE Conference on Intelligent Robots, pages 578-583. SPIE, November, 1983.

[Henderson 83a] Henderson, Thomas C. and Wu So Fai.
A Multi-sensor Integration and Data Acquisition System.
In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 274-280. IEEE, June, 1983.

[Henderson 83b] Henderson, T.C. and Wu So Fai.
Pattern Recognition in a Multi-sensor Environment.
UUCS 001, University of Utah, July, 1983.

[Henderson 83c] Henderson, T.C., E. Shilcrat and C. Hansen.
A Fault Tolerant Sensor Scheme.
Computer Science UUCS 83-003, University of Utah, November, 1983.

[Jacobsen 83] Jacobsen, S., J.E. Wood, D.F. Knutti and K. Biggers.
The Utah/MIT Dextrous Hand.
In MIT/SDF IRR Symposium. August, 1983.

[Keller 77] Keller R. M.
Semantics of Parallel Program Graphs.
Department of Computer Science UUCS-77-110, University of Utah, July, 1977.

[Keller 82] Keller R.M.
FEL Programmer's Guide.
AMPS Technical Memorandum 7, Univ. of Utah; Dept. of Computer Science, April, 1982.

[Keller 83] Keller R. M., Lindstrom G., Organick E. I.
REDIFLOW: A Multiprocessing Architecture Combining Reduction and Data-Flow.
In Parallel Architecture Workshop. Univ. of Colorado, 1983.

[Randell 77] Randell, B.
System Structure for Software Fault Tolerance.
Prentice-Hall, Englewood Cliffs, NJ, 1977, pages 195-219.

[Trelevan 82] Trelevan P. C., Brownbridge D. R., Hopkins R. P.
Data-Driven and Demand-Driven Computer Architecture.
ACM Computing Surveys 14(1):93-143, March, 1982.

[Wu 83] Wu So Fai.
A Multi-sensor Integration and Data Acquisition System.
Master's thesis, University of Utah, June, 1983.