Instrumented Sensor System Architecture

Mohamed Dekhil and Thomas C. Henderson

UUSC-97-011

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

November 1997

Abstract

Sensor systems are becoming ubiquitous throughout society, yet their design, construction and operation are still more of an art than a science. In this paper, we define, develop, and apply a formal semantics for sensor systems that provides a theoretical framework for an integrated software architecture for modeling sensor-based control systems. Our goal is to develop a design framework which allows the user to model, analyze and experiment with different versions of a sensor system. This includes the ability to build and modify multisensor systems and to monitor and debug both the output of the system and the affect of any modification in terms of robustness, efficiency, and error measures. The notion of *Instrumented Logical Sensor Systems* (ILSS) that are derived from this modeling and design methodology is introduced. The instrumented sensor approach is based on a sensori-computational model which defines the components of the sensor system in terms of their functionality, accuracy, robustness and efficiency. This approach provides a uniform specification language to define sensor systems as a composition of smaller, predefined components. From a software engineering standpoint, this addresses the issues of modularity, reusability, and reliability for building complex systems. An example is given which compares vision and sonar techniques for the recovery of wall pose.

This work was supported in part by NSF grant CDA 9024721 and a gift from Hewlett Packard Corporation.

1 Introduction

In any closed-loop control system, sensors are used to provide the feedback information that represents the current status of the system and the environmental uncertainties. Building a sensor system for a certain application is a process that includes the analysis of the system requirements, a model of the environment, the determination of system behavior under different conditions, and the selection of suitable sensors. The next step in building the sensor system is to assemble the hardware components and to develop the necessary software modules for data fusion and interpretation. Finally, the system is tested and the performance is analyzed. Once the system is built, it is difficult to monitor the different components of the system for the purpose of testing, debugging and analysis. It is also hard to evaluate the system in terms of time complexity, space complexity, robustness, and efficiency, since this requires quantitative measures for each of these measures.

In addition, designing and implementing real-time systems are becoming increasingly complex because of many added features such as fancy graphical users interfaces (GUIs), visualization capabilities and the use of many sensors of different types. Therefore, many software engineering issues such as reusability and the use of COTS (Commercial Off-The Shelf) components [31], real-time issues [34, 33, 23], sensor selection [11], reliability [26, 27, 35], and embedded testing [36] are now getting more attention from system developers.

In a previous paper, we proposed to use formal semantics to define performance characteristics of sensor systems [4]. In this paper, we address these and other problems related to sensor system modeling and evaluation. We start by presenting a theoretical framework for modeling and designing sensor systems based on a formal semantics in terms of a virtual sensing machine. This framework defines an explicit tie between the specification, robustness and efficiency of the sensor system by defining several quantitative measures that characterize certain aspects of the system's behavior. Figure 1 illustrates our proposed approach which provides static analysis (e.g., time/space complexity, error analysis) and dynamic handles that assist in monitoring and debugging the system.

1.1 Sensor Modeling

Each sensor type has different characteristics and functional description. Therefore it is desirable to find a general model for these different types that allows modeling sensor systems that are independent of the physical sensors used, and enables studying the performance and robustness of such systems. There have been many attempts to provide "the" general model along with its mathematical basis and description. Some of these modeling techniques concern error analysis and fault tolerance of multisensor systems [29, 3, 28, 24, 30, 6]. Other techniques are model-based and require a priori knowledge of the scanned object and its environment [8, 13, 25]. These techniques help fit data to a model, but do not provide the means to compare alternatives. Task-directed

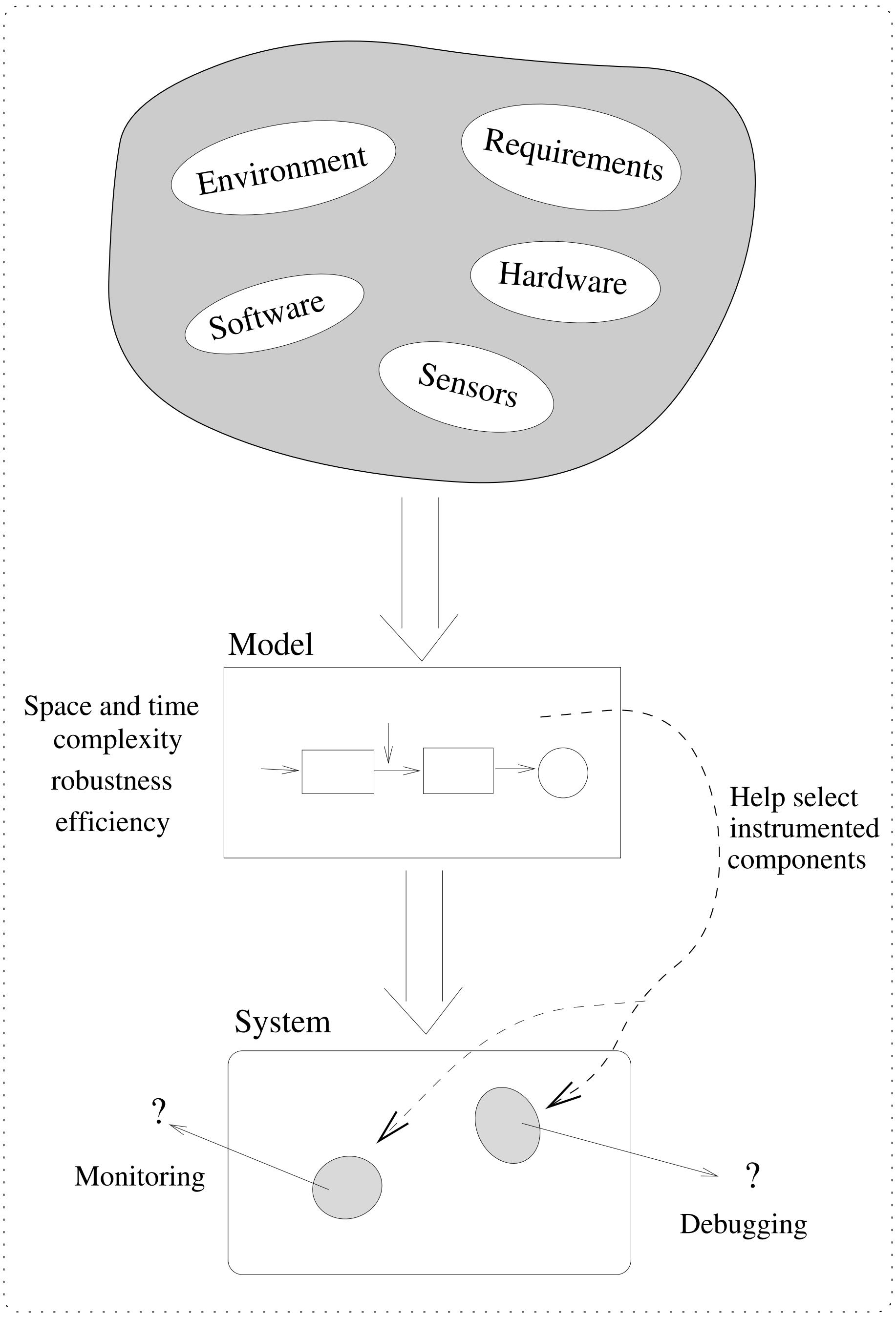

Figure 1: The proposed modeling approach.

sensing is another approach to devise sensing strategies [15, 14, 2], but again, it does not provide measures to evaluate the sensor system in terms of robustness and efficiency.

Another approach to modeling sensor systems is to define sensori-computational systems associated with each sensor to allow design, comparison, transformation, and reduction of any sensory system [7]. In this approach the concept of information invariants is used to define some measure of information complexity. This approach provides a very strong computational theory which allows comparing sensor systems, reducing one sensor system to another, and measuring the information complexity required to perform a certain task. However, as stated by Donald, the measures for information complexity are fundamentally different from performance measures. Also, this approach does not permit one to judge which system is "simpler," "better," or "cheaper."

To that end, we introduce the notion of an *Instrumented Logical Sensor System* (ILSS) which represents our methodology for incorporating design tools and allows static and dynamic performance analysis, on-line monitoring, and embedded testing. Figure 2 shows the components of our framework. First (on the left), an Instrumented Logical Sensor Specification is defined, as well as $\mathcal{F}$, a set of functions which measure system properties of interest. This specification is derived from a mathematical model, simulation results, or from descriptions of system components. Analysis of some aspects of the ILSS are possible (e.g., worst-case complexity of algorithms). Next (the center of the figure), an implementation of the system is created; this can be done by hand or automatically generated in a compile step (note that the original Logical Sensor Specifications[21] could be compiled into Unix shell script or Function Equation Language (FEL), an applicative language). Either way, the monitoring, embedded testing or taps are incorporated into the system implementation. Finally (the right hand side), validation is achieved by analyzing the system response and performance measures generated during system execution. In this way, there are some semantic constraints on the values monitored which relate the system output measures to the original question posed for the specification.

Currently, an ILSS library is under development as part of an interactive graphical programming environment called "CWave" used to design and execute real-time control systems.[1] Currently, we have a theoretical framework and validation strategy with a partial implementation within CWAVE. CWave is a graphical program specification language that has been created to design measurement systems and has been funded by HP. CWave has been applied to broad robot systems (e.g., Lego robot warehouse demos) in our software engineering projects class here at Utah. Finally, CWave is a specification language and can be linked to simulation tools, or executed in an interpreted mode, or compiled for incorporation in embedded systems.

[1] refer to "http://easy.cs.utah.edu/cwave/index.htm" for more information about the CWave project.

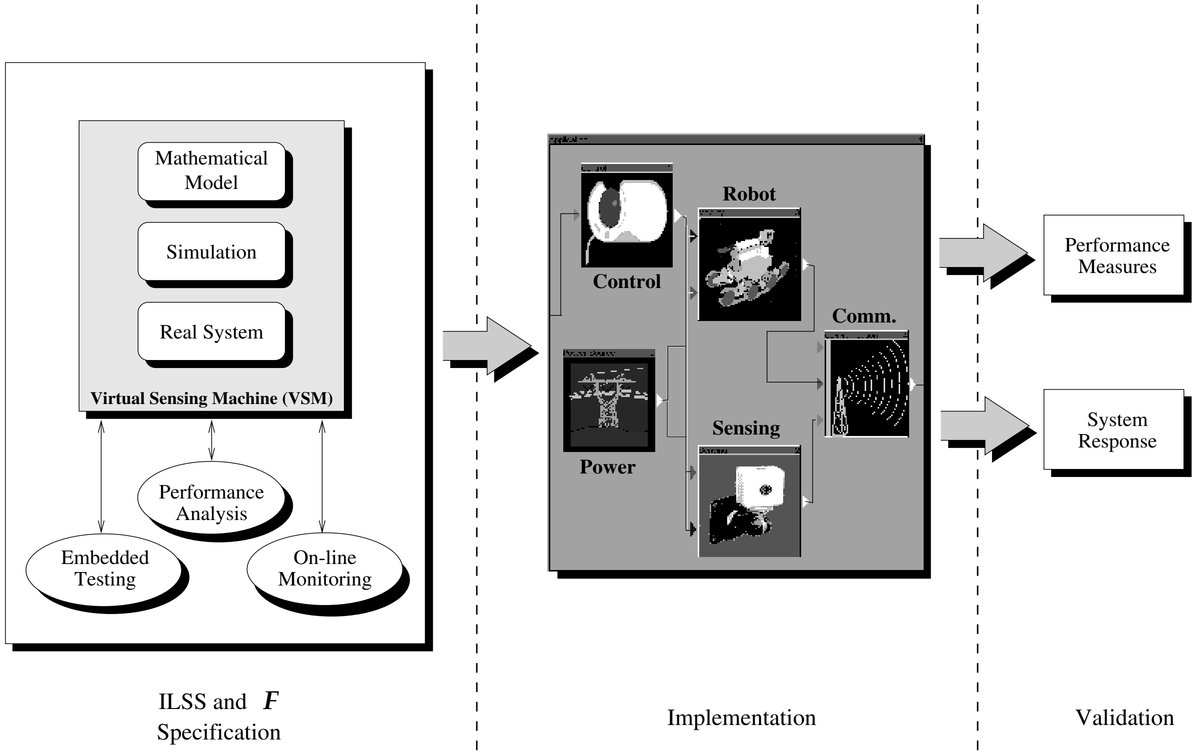

Figure 2: The Instrumented Logical Sensor System Components.

2 Performance Semantics of Sensor Systems

The use of sensors in safety critical applications, such as transportation and medicine, requires a high level of reliability. However, increased robustness and reliability of a multisensor system requires increased cost through redundant components and more sensor readings and computation. In contrast, increasing the efficiency of the system means less redundant components, fewer sensor readings and less computation. Performance analysis is crucial to making an informed tradeoff between design alternatives.

Performance analysis consists of a static analysis of a specification of the system and its parameters as well as a dynamic analysis of the system's run-time behavior. The static analysis can be based on some formal description of the syntax and semantics of the sensor system, while the dynamic analysis requires on-line monitoring of some quantitative measures during run-time.

Our goal is to achieve strong performance analysis and provide information which allows the user to make informed choices concerning system tradeoffs. This involves a sensor system model which permits quantitative measures of time and space complexity, error, robustness, and efficiency, and which facilitates analysis, debugging and on-line monitoring.

Formal semantics of programming languages provides techniques to describe the meaning of a language based on precise mathematical principles. These formal techniques should provide the following: precise machine-independent concepts, unambiguous specification techniques, and a rigorous theory to support reliable reasoning [12]. The main types of formal semantics are: *denotational semantics* which concerns designing denotations for constructs, *operational semantics* which concerns the specification of an abstract machine together with the machine behavior when running the program, and *axiomatic semantics* which concerns axioms and rules of inference for reasoning about programs.

Our view is that performance semantics should allow us to compute measures of interest on program structures. Denotational semantics is the closest to our view since, according to [1], to specify the semantics of a language denotationally means to specify a group of functions which assigns mathematical objects to the program and to parts of programs (modules) in such a way that the semantics of a module depends only on the semantics of the submodules. Thus, given a set of programs, $\mathcal{P}$, from a language, and an operating context, $\mathcal{C}$, the semantics is a set of functions

$$\mathcal{F} = \{f_i\}$$

where

$$f_i : \mathcal{P} \times \mathcal{C} \rightarrow \Re$$

where $\Re$ is the measurement domain.

The static semantics defines structural measures over the syntax of $p \in \mathcal{P}$. This includes standard measures such as maximum depth of the program graph, branching measures, data structure properties, storage estimates and standard computational complexity measures. Note that these can be determined without reference to $\mathcal{C}$ (i.e., $f : \mathcal{P} \rightarrow \Re$). This can be extended to include functions of the operational context $\mathcal{C}$, including sensor models, accuracy, precision, redundancy and replacement, as well as operating system effects, communication strategies and protocols, and processor properties.

The dynamic semantics include validity measures and operational characteristics. Validity measures permit the comparison of behavior models to actual run-time performance (monitors), while operational characteristics are simply measures of run-time values (taps). The values of a tap or monitor are represented as a sequence $X = (x_n : n \in \mathcal{N})$; x_n is the n^{th} value produced by the tap or monitor

$$X : \mathcal{N} \rightarrow S$$

where S is the structure produced by the tap or monitor.

The selection of functions in $\mathcal{F}$ depends directly on the user's needs and are defined so as to answer specific questions. Standard questions include actual running times, space requirements, bottlenecks, etc., and a complex application can be investigated in a top down manner – the user may define new measurement functions on lower level modules once information is gained at a higher level. This forces the user to identify crucial parameters and to measure their impact. For example, a computer vision application may be data dependent, say on the number of segmented objects or their distribution in the image. Thus, the user is coerced into a better understanding of the significant value regimes of these parameters and may develop monitors to ensure that the application stays within a given range, or that it dynamically switches algorithms when a particular parameter value occurs (e.g., more than 1000 segmented objects occur in the image). The main point is that the user can construct executable versions of the $f_i \in \mathcal{F}$ to ensure the validity of the controller as it runs.

Although computational complexity provides insight for worst case analysis, and for appropriate population distribution models, average case analysis can be performed, we propose here what might be termed *empirical case analysis* which allows the user to gain insight into the system without requiring a detailed analytical model of the entire application and its context. Very few users exploit formal complexity analysis methods; we believe that empirical case analysis is a very useful tool.

2.1 Simple Example: Time Vs. Robustness Using Sonar Readings

Suppose that we want to determine how many sonar readings to use to get a robust range estimate, but would like to trade off against the time taken to sample. This simple example demonstrates the motivation of the proposed approach and how it can be used to select between alternatives. In this example we have a "classical" tradeoff between speed (time to accomplish a certain task) and robustness (a combination of accuracy and repeatability). Assume that the sonar has been calibrated to eliminate any environmental effects (e.g., wall type, audio noises, etc.). The variables in this case are the accuracy of the physical sonar sensor and the number of readings taken for the same position.

Assuming the time to take one reading is t, the error standard deviation is σ, and the probability of a bad reading is Pr_b, taking one reading yields minimum time and worst accuracy. By adding a filter (e.g., averaging) and taking multiple readings, accuracy increases and time also increases. Therefore, we need quantitative measures to decide how many readings are needed to achieve the required accuracy (measured in terms of the standard deviation of the error) within a time limit.

Using the formalism presented earlier, the semantics of this problem can be defined using the set of functions $\mathcal{F} = \{time, error, repeatability\}$. In the case of using a single reading these functions can be written as:

$$time(single) = t$$

$$error(single) = \frac{\sigma}{\sqrt{(1 - Pr_b)}}$$

$$repeatability(single) = 1 - Pr_b$$

Now, if we take the average of n readings, the semantics can be written as:

$$time(average) = nt + \tau_n$$

$$error(average) = \frac{\sigma}{\sqrt{n * (1 - Pr_b)}}$$

$$repeatability(average) = 1 - Pr_b^n$$

where τ_n is the time to calculate the average of n readings, and $\tau_1 = 0$.

In this simple example we were able to get estimates of the required measures using mathematical models. However, we did not consider the changes in the environment and how it affects these measures. In this case, the set of functions $\mathcal{F}$ are mappings from the cross product of the program $\mathcal{P}$ and the operating context C to the measurement domain $\Re$, that is

$$f_i : \mathcal{P} \times \mathcal{C} \rightarrow \Re$$

To solve this problem, we either have to model the environmental effects and include it in our model, or we may need to conduct simulations if a mathematical model is not possible. Simulation is a very useful tool to approximate reality, however, in some cases even simulation is not enough to capture all the variables in the model, and real experiments with statistical analysis may be required to get more accurate results. Thus, the formal functions can be operationalized as monitors or taps in the actual system.

3 Sensor System Specification

The ILSS approach is based on *Logical Sensor Systems* (LSS) introduced by Henderson and Shilcrat [21]. LSS is a methodology to specify any sensor in such a way that hides its physical nature. The main goal behind LSS was to develop a coherent and efficient presentation of the information provided by many sensors of different types. This representation provides a means for recovery from sensor failure and also facilitates reconfiguration of the sensor system when adding or replacing sensors [20].

We define the ILSS as an extension to the LSS and it is comprised of the following components (see Figure 3):

1. *ILS Name*: uniquely identifies a module.

2. *Characteristic Output Vector (COV)*: strongly typed output structure. We have one output vector (COV_{out}) and zero or more input vectors (COV_{in}).

3. *Commands*: input commands to the module ($Commands_{in}$) and output commands to other modules ($Commands_{out}$).

4. *Select Function*: selector which detects the failure of an alternate and switches to another alternate (if possible).

5. *Alternate Subnets*: alternative ways of producing the COV_{out}. It is these implementations of one or more algorithms that carry the main functions of the module.

6. *Control Command Interpreter (CCI)*: interpreter of the commands to the module.

7. *Embedded Tests*: self testing routines which increase robustness and facilitate debugging.

8. *Monitors*: modules that check the validity of the resulting COVs.

9. *Taps*: hooks on the output lines to view different COV values.

These components identify the system behavior and provide mechanisms for on-line monitoring and debugging. In addition, they give handles for measuring the run-time performance of the system.

Monitors are validity check stations that filter the output and alert the user to any undesired results. Each monitor is equipped with a set of rules (or constraints) that governs the behavior of the COV under different situations.

Embedded testing is used for on-line checking and debugging proposes. Weller proposed a sensor processing model with the ability to detect measurement errors and to recover from these errors [36]. This method is based on providing each system module with verification tests to verify certain characteristics in the measured data and to verify the internal and output data resulting from the sensor module algorithm. The recovery strategy is based on rules that are local to the different sensor modules. We use a similar approach in our framework called *local embedded testing* in which each module is equipped with a set of tests based on the semantic definition of that module. These tests generate input data to check different aspects of the module, then examine the output of the module using a set of constraints and rules defined by the semantics. Also these tests can take input data from other modules if we want to check the operation for a group of modules.

Figure 4 illustrates the idea of local embedded testing. Local embedded testing increases the robustness of the system and provides the user with possible locations to tap into when there is a problem with the system.

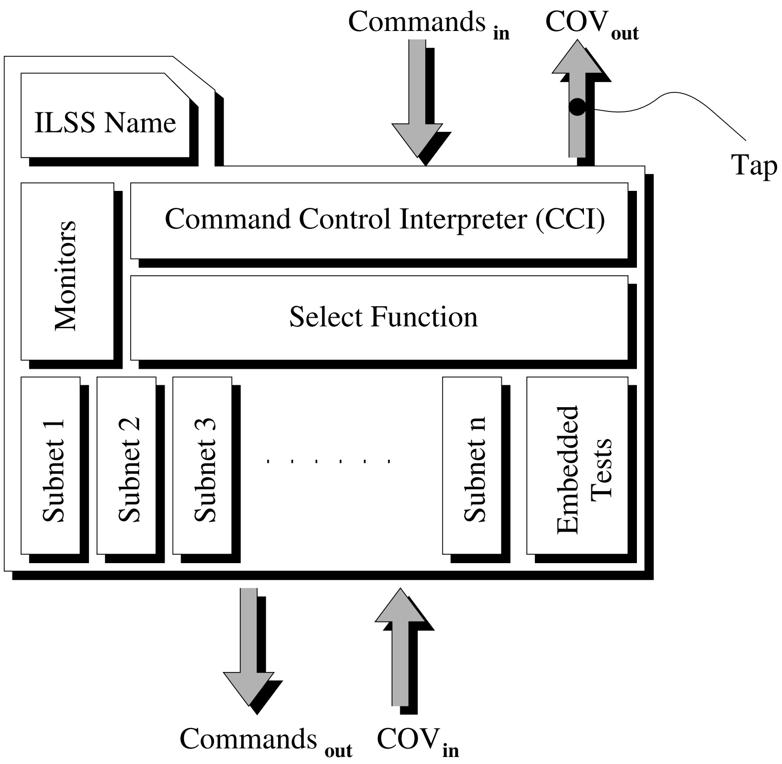

Figure 3: The extended logical sensor module.

3.1 Construction Operators

In our proposed framework, a sensor system is composed of several ILSS modules connected together in a certain structure. We define operations for composing ILSS modules, and then define the semantics of these operations in terms of the performance parameters. Some of these operations are (see Figure 5):

- $Serial(ILSS1, ILSS2)$: two logical modules are connected in series. Here $COV3 = COV2$.
- $Select(ILSS1, ILSS2)$: $COV3$ is equal to either $COV1$ or $COV2$.
- $Combine(ILSS1, ILSS2)$: $COV3$ is the concatenation of $COV1$ and $COV2$.

For these simple constructs, the semantics is defined as a set of functions that propagate the required performance measures. Several techniques can be used for propagation. Best case analysis, worst case analysis, average, etc. Selecting among these depends on the application, hence it

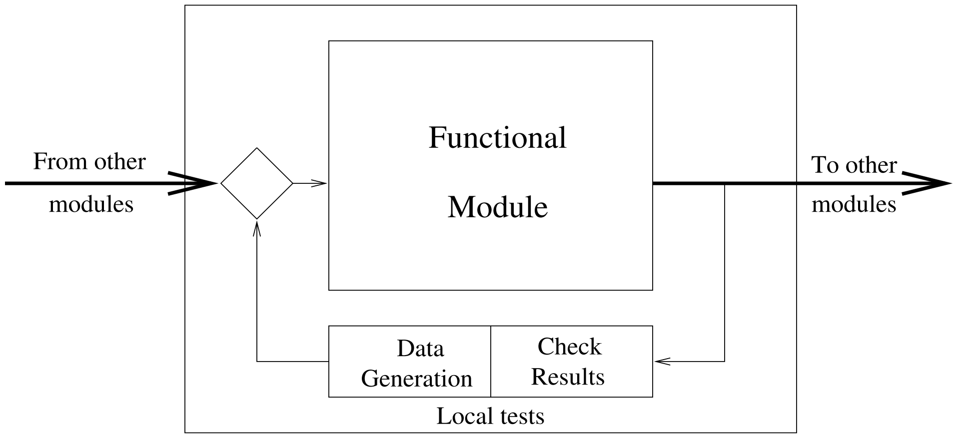

Figure 4: Local embedded testing.

should be user defined. As an example, the time of the resulting logical system using worst case analysis can be calculated as follows:

- $time(Serial(ILSS1, ILSS2)) = time(ILSS1) + time(ILSS2)$
- $time(Select(ILSS1, ILSS2) = max(time(ILSS1), time(ILSS2))$
- $time(Combine(ILSS1, ILSS2) = max(time(ILSS1), time(ILSS2))$

Hence, the semantic functions of the composite system are defined in terms of the semantic functions of the subcomponents, Similarly, functions that define the propagation of other performance measures can be defined in the same way.

For error propagation, we use a simple approach which does not require carrying a lot of information through the system. This approach is based on the uncertainty propagation described in [22, 9]. Assume that we have a certain module with n inputs $X = (x_1, x_2, \ldots, x_n)$ and m outputs $Y = (y_1, y_2, \ldots, y_m)$ such that $Y = f(X)$, and assume that the error variance associated with the input vector is $\Lambda_X = (\Lambda_{x_1}, \Lambda_{x_2}, \ldots, \Lambda_{x_n})$ (see Figure 6), then the error variance for the output vector is calculated using the equation:

$$\Lambda_Y = \left(\frac{\partial Y}{\partial X}\right) \Lambda_X \left(\frac{\partial Y}{\partial X}\right)^T$$

where $\frac{\partial Y}{\partial X}$ is the partial derivative of Y with respect to X evaluated at the measured value of the input vector X. If all the elements in X are independent variables, then this equation can be written

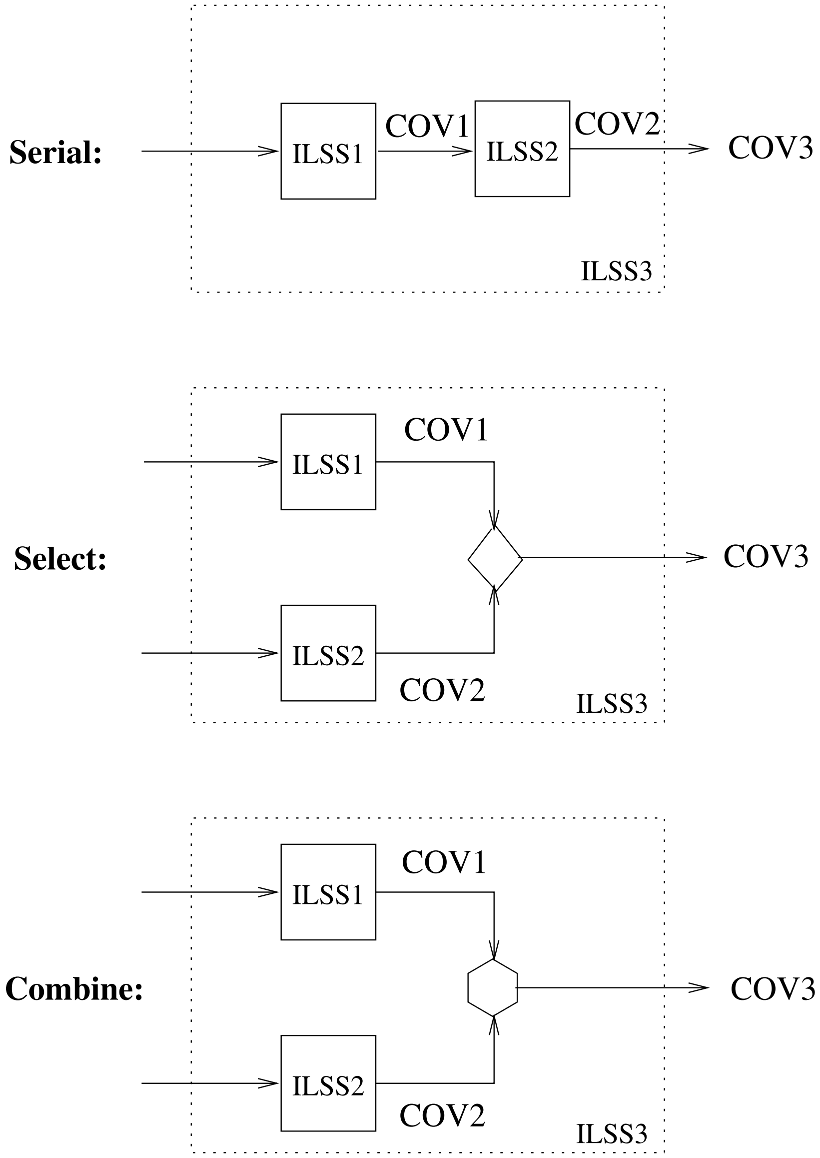

Figure 5: Some operations used for propagating the performance measures.

as:

$$\Lambda_{y_i} = \sum_{j=1}^{n} \left(\frac{\partial y_i}{\partial x_j} \right)^2 \Lambda_{xj} \text{ , } i = 1, 2, \ldots, m$$

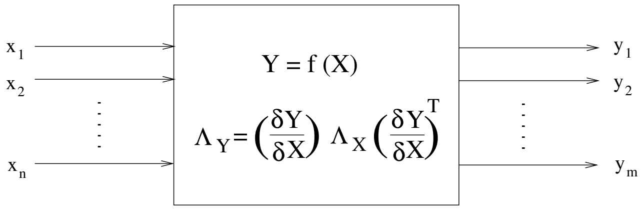

Figure 6: A simple approach for error propagation.

Our overall goal is to provide a tightly coupled mechanism to map high-level performance measures onto an appropriate set of monitors, tests and taps so as to provide the required information.

4 Implementation

The ultimate goal of this project is to utilize the proposed theoretical framework in a usable modeling and prototyping environment with tools for analysis, debugging, and monitoring sensor systems with emphasis on robot control applications. Thus, we are developing an ILSS library within a visual programming system called CWave targeted toward the development of control systems for measurement devices and hardware simulations. CWave is developed by the Component Software Project (CSP) research group in the Department of Computer Science at the University of Utah in cooperation with the CSP group at Hewlett Packard Research Labs in Palo Alto, California.

CWave is based on a reusable software components methodology where any system can be implemented by visually wiring together predefined and/or user created components and defining the dataflow between these components. The CWave design environment includes several important features that make it suitable to use as a framework for implementing ILSS components. Some of these features are:

- Open architecture with ease of extensibility.
- Drag-and-drop interface for selecting components.
- Several execution modes including single step, slow, and fast execution.
- On-line modification of component properties.

- The ability to add code interactively using one of several scripting languages including Visual Basic and Java Script. This is particularly useful to add monitors and/or taps on the fly.
- Parallel execution using visual threads.
- On-line context sensitive help.

Figure 7 shows the CWave design environment with some of its features.

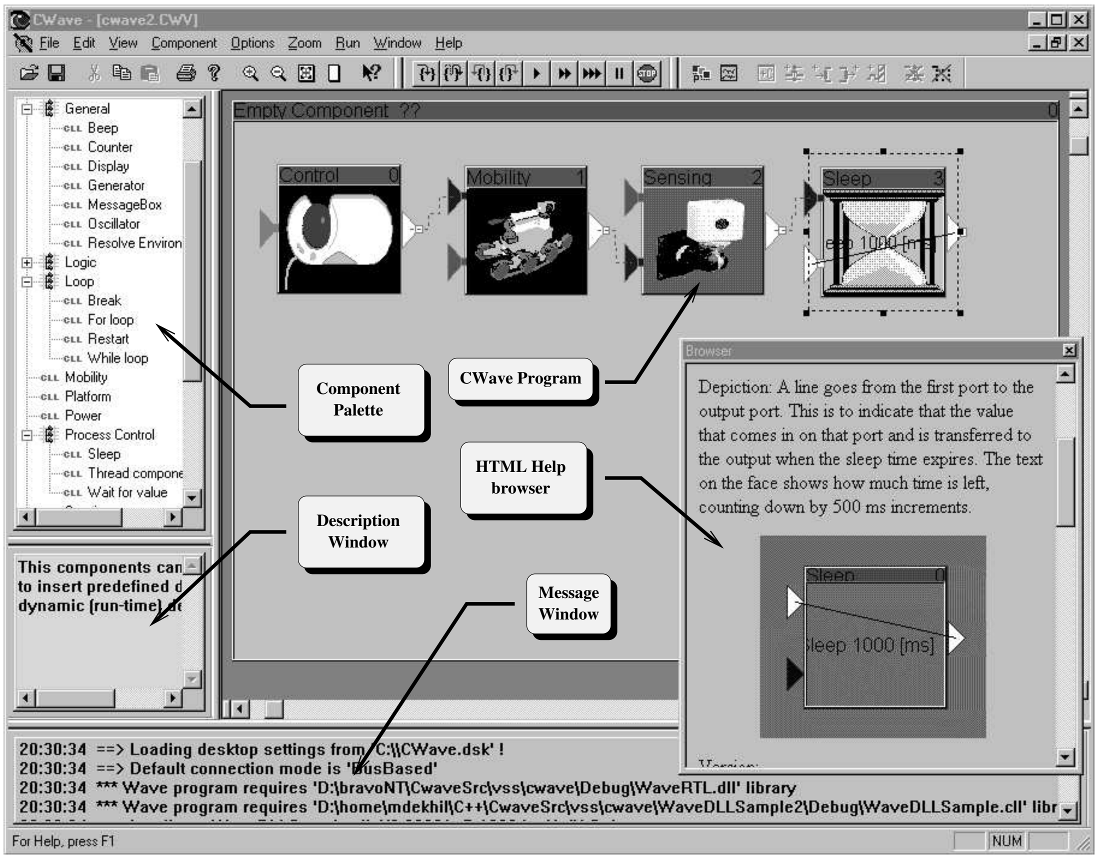

Figure 7: CWave design environment.

An object-oriented approach is used to develop the ILSS components using Visual C++ for implementation. Each component is an object that possesses some basic features common to all components plus some additional features that are specific to each ILSS type. The following are some of the basic functions supported by all components:

Initialize: performs some initialization steps when the component is created.

Calibrate: starts a calibration routine.

Sense: generates the COV corresponding to the current input and the component status.

Reset: resets all the dynamic parameters of the component to their initial state.

Test: performs one or more of the component's embedded tests.

Select: selects one of the alternate subnets. This allows for dynamic reconfiguration of the system.

Monitor: observes the COV and validate its behavior against some predefined characteristic criteria.

Tap: displays the value of the required variables.

We used several design patterns in designing and implementing the components. Design patterns provide reliable and flexible object-oriented designs that can accommodate rapid modifications and extensions [10]. For example, the *decorator* pattern is used to dynamically attach additional functionality to the object. This is particularly useful in our case where the user can dynamically choose the performance measures to be propagated and the values to be monitored while the system is running. Note that monitors, tests, and taps can be exploited to analyze CWave (or any implementation language) module performance independently of the sensor aspects of the system. This is rendered more efficient and transparent to the user by incorporating them directly as language features.

5 Example: Wall Pose Estimation

The following example illustrates the use of the proposed framework to model and analyze two alternatives for determining flat wall position and orientation: one using vision and one using sonar sensors [5, 16, 18, 19]. The sonar sensors are mounted on a LABMATE mobile robot designed by Transitions Research Corporation. The LABMATE was used for several experiments in the Department of Computer Science at the University of Utah. It was also entered in the 1994 and 1996 AAAI Robot Competition [32] and it won sixth and third place, respectively. For that purpose, the LABMATE was equipped with 24 sonar sensors, eight infrared sensors, a camera and a speaker. [2] Figure 8 shows the LABMATE with its equipment.

[2]The LABMATE preparations, the sensory equipments, and the software and hardware controllers were done by L. Schenkat and L. Veigel at the Department of Computer Science, University of Utah.

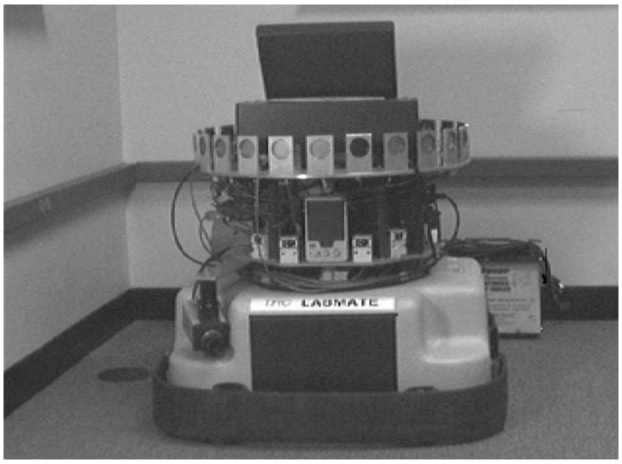

Figure 8: The LABMATE robot with its equipment.

In this example, we consider two different logical sensors to determine wall pose and find the corresponding errors and time complexity for each. The first ILSS uses a camera and known target size and location. The second ILSS deals with the sonar sensor as a wedge sensor (i.e., it returns a wedge centered at the sonar sensor and spread by an angle 2θ.) Figure 9 shows the two logical sensors. (See [18] for an overview of the sonar pose recovery technique, and [17] for target-based calibration.)

In this figure, *image* is the 128x128 black and white image acquired by the *Camera*, and r_1 and r_2 are the two sonar readings generated from $Sonar1$ and $Sonar2$, respectively. *Target Points* extracts three reference points from the *image*, while *Vision Line* produces two points on the line of intersection of the wall with the x-z plane of the camera system. $Wedge\_Sonar\_Line$ takes the two range values r_1 and r_2, and the spread angle of the sonar beam θ, and returns two 2D points on the line representing the wall.

5.1 System Modeling and Specification

As shown in Figure 9, ILSS1 is composed of three modules, a *Camera* module, a *Target Points* module and a *Vision Line* module. On the other hand, LSS2 has three modules, two *Sonar* modules and a *Wedge_Sonar_Line* module followed by a $Combine$ operator.

Each ILSS is defined in terms of a set of components that characterize the module. The data and the corresponding performance measures start from the $Camera$ or $Sonar$ module and propagate upward until they reach the COV of the main ILSS. On the other hand, the commands start from the main ILSS and propagate downward until they reach the $Camera$ or $Sonar$ module. The COV

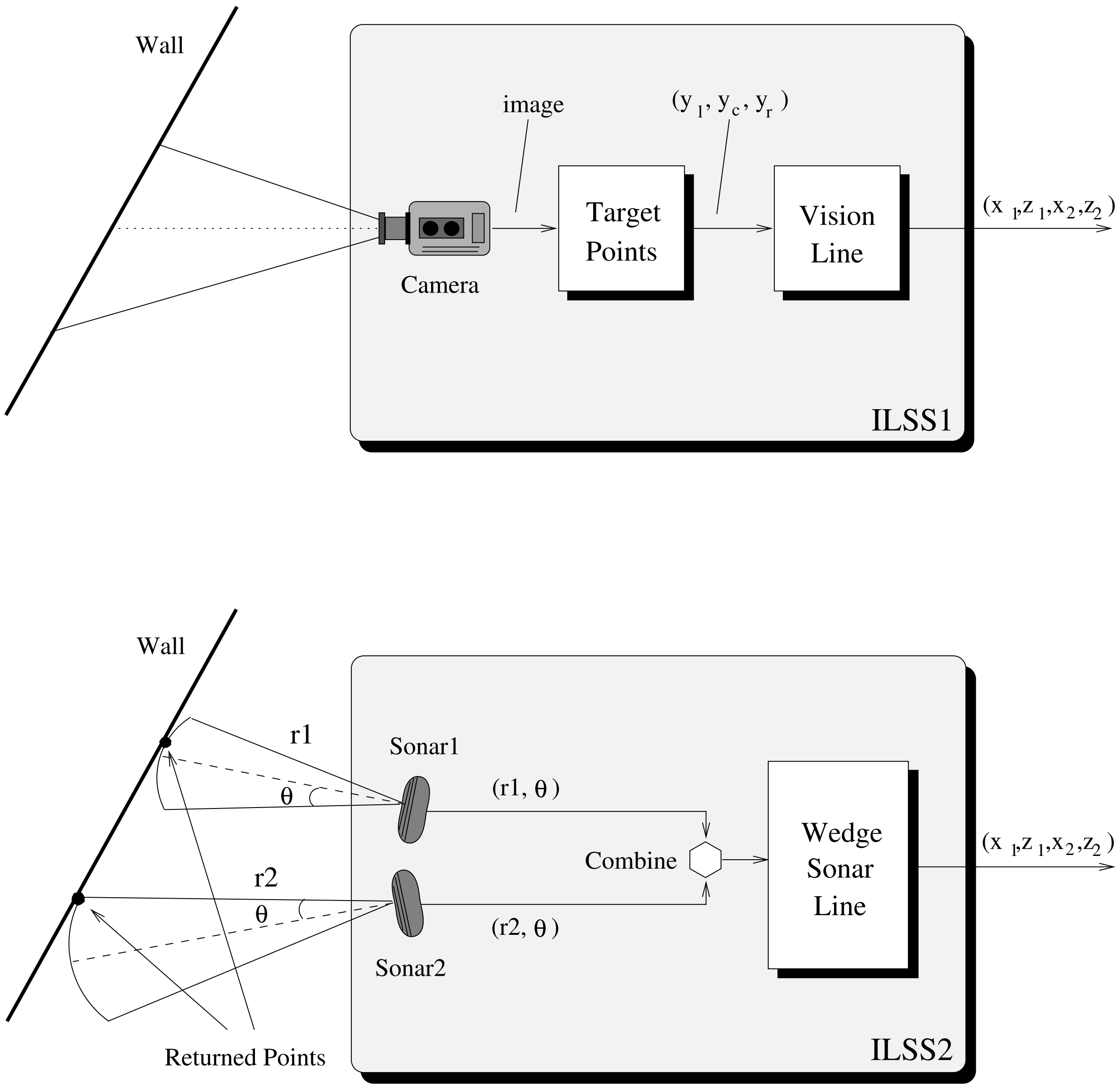

Figure 9: Two Instrumented Logical Sensors for determining wall position.

is composed of two parts: *data* and *performance measures*. For example, COV_{out} for $Sonar1$ is

$$(\{r_1, \theta\}, \{t, \Lambda_{r1}, \Lambda_\theta\})$$

where t is the time taken to execute the module and Λ_{r1} and Λ_θ are the error variances for r_1 and θ, respectively. In this example, each module has only one alternate subnet, therefore, the select function is trivial.

5.2 Performance Semantic Equations

Using worst case analysis, the performance semantic equations of the *time* and *error* for ILSS1 and ILSS2 can be written as:

$$time(ILSS1) = time(Serial(Camera, TargetPoints, VisionLine))$$

$$error(ILSS1) = error(Serial(Camera, TargetPoints, VisionLine))$$

$$time(ILSS2) = time(serial(combine(Sonar1, Sonar2), Wedge\_sonar\_line))$$

$$error(ILSS2) = error(serial(combine(Sonar1, Sonar2), Wedge\_sonar\_line))$$

Now, we need to calculate the time and error for the subcomponents. Assume that t_{sonar1}, t_{sonar2}, t_{camera}, $t_{TargetPoints}$, $t_{VisionLine}$ and $t_{wedge\_sonar\_line}$ are the time for the subcomponents, and Λ_{r1}, Λ_{r2}, Λ_{y_l}, Λ_{y_c}, Λ_{y_r} and Λ_θ are the error measures for r_1, r_2, y_l, y_c, y_r and θ, respectively. The time for LSS1 and LSS2 can be easily calculated using the propagation operations discussed earlier as follows:

$$time(ILSS1) = t_{camera} + t_{TargetPoints} + t_{VisionLine}$$

$$time(ILSS2) = max(t_{sonar1}, t_{sonar2}) + t_{wedge\_sonar\_line}$$

Propagating the error requires more elaborate analysis for each component. For ILSS1, we start with the error in the physical sensor which is the camera in this case. The camera generates two-dimensional arrays of intensity values, $P(x, y)$, where P is an $m \times n$ matrix. The error we are concerned abound in this example is the error in position (x, y) of a point on the CCD array (which corresponds to rows and columns in the image.) This error is affected by the resolution of the camera and the distance between the CCD elements. Let's assume that the error is Gaussian with mean 0 and variance (Λ_x, Λ_y) at any point (x, y). This can be written as:

$$error(Camera) = \{(\Lambda_x, \Lambda_y)_{m\times n}\}$$

This error translates directly into the second component, $Target\_Points$, which extracts the y value for three different points in the image; y_l, y_c, and y_r. Assuming that the variance in the y direction (Λ_y) is the same at any pixel, the error at this stage will be:

$$error(Target\_Points) = \{\Lambda_y, \Lambda_y, \Lambda_y\}$$

The last component in ILSS1, $Vision\_Line$ performs several operations on these three values to generate the two points of the line representing the wall. First, the corresponding z value is calculated for the three points using the equation:

$$z_i = \frac{Y_0}{y_i}, \qquad i = l, c, r$$

where Y_0 is the height of the physical point and is a known constant in our example. The error associated with z_i can be calculated as follows:

$$\Lambda_{z_i} = \left(\frac{\partial z_i}{\partial y_i}\right)^2 \Lambda_{y_i}$$

By calculating the derivative in the above equation we get:

$$\Lambda_{z_i} = \left(\frac{-Y_0}{y_i^2}\right)^2 \Lambda_y = \frac{Y_0^2}{y_i^4}\Lambda_y$$

which shows how Λ_{z_i} depends on the value of y_i. Second, the angle between the robot and the wall (α) is calculated with the function:

$$\alpha = sin^{-1}\left(\frac{z_l - z_r}{D_0}\right)$$

where D_0 is the known distance between the two physical points p_l and p_r. Therefore,

$$\Lambda_\alpha = \left(\frac{\partial \alpha}{\partial z_l}\right)^2 \Lambda_{z_l} + \left(\frac{\partial \alpha}{\partial z_r}\right)^2 \Lambda_{z_r}$$

$$= \left(\frac{1}{\sqrt{1 - \left(\frac{z_l - z_r}{D_0}\right)^2}}\right)^2 \Lambda_{z_l} + \left(\frac{-1}{\sqrt{1 - \left(\frac{z_l - z_r}{D_0}\right)^2}}\right)^2 \Lambda_{z_r}$$

After simplifying the last equation we get:

$$\Lambda_\alpha = \frac{D_0^2}{D_0^2 - (z_l - z_r)^2}(\Lambda_{z_l} + \Lambda_{z_r})$$

Finally, we calculate two points on the line representing the wall as shown in Figure 10. Take the first point p_1 at $(0, z_c)$ and the second point p_2 at one unit distance from p_1 along the wall which gives the point $(\cos\alpha, z_c + \sin\alpha)$:

$$x_1 = 0, \quad z_1 = z_c$$

$$x_2 = \cos\alpha, \quad z_2 = z_c + \sin\alpha$$

From these equations, the error for the two points will be:

$$\Lambda_{x_1} = 0, \quad \Lambda_{z_1} = \Lambda_{z_c}$$

$$\Lambda_{x_2} = sin^2\alpha\ \Lambda_\alpha, \quad \Lambda_{z_2} = \Lambda_{z_c} + cos^2\alpha\ \Lambda_\alpha$$

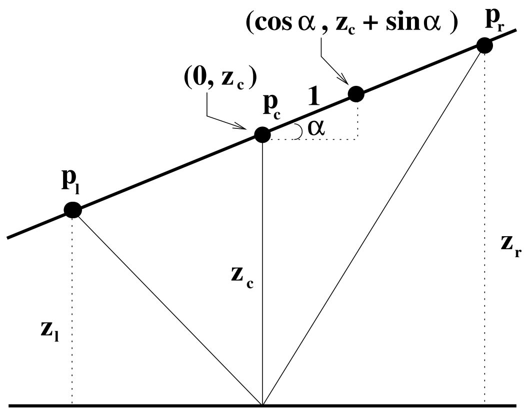

Figure 10: The two points on the line representing the wall

Now, we can write the error of ILSS1 as:

$$error(ILSS1) = \{\Lambda_{x_1}, \Lambda_{z_1}, \Lambda_{x_2}, \Lambda_{z_2}\}$$

Notice that we can write the error in terms of $\Lambda_y, Y_0, D_0, y_l, y_c$, and y_r. For example, let's assume that $\Lambda_y = 1mm^2, Y_0 = 500mm, D_0 = 300mm$, and $y_l = y_c = y_r = 10mm$ (α is zero in this case), then the error will be:

$$error(ILSS1) = \{0, 25mm^2, 0, 25mm^2\}$$

Now we analyze ILSS2 in a similar manner. At the first level, we have the physical sonar sensor where the error can be determined either from the manufacturer specs, or from experimental data. In this example we will use the error analysis done by Schenkat and Veigel [32] in which there is a Gaussian error with mean μ and variance σ^2. From this analysis, the variance is a function of the returned distance r. To simplify the problem let's assume that the variance in both sensors is $\Lambda_r = 4.0mm^2$. Therefore we can write the error in the sonars as:

$$error(Sonar) = \{\Lambda_r\}$$

In the $Wedge\_Sonar\_Line$ module, there are five possible cases for that line depending on the values of r_1 and r_2 [18]. In any case, the two points on the line can be written as:

$$x_1 = r_1 \cos \alpha_1, \quad z_1 = r_1 \sin \alpha_1$$

$$x_2 = r_2 \cos \alpha_2, \quad z_2 = r_2 \sin \alpha_2$$

where the values of $\alpha 1$ and α_2 are between $-\theta$ to θ (see Figure 11).

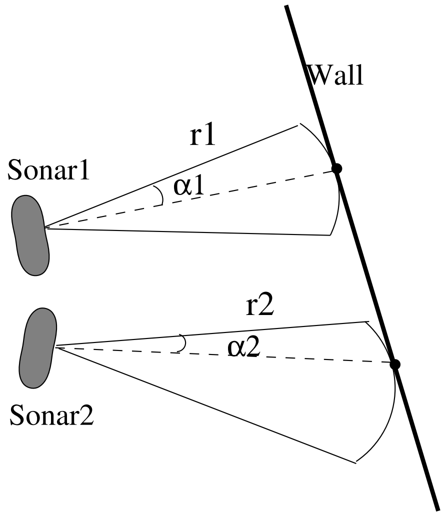

Figure 11: The general case for the points returned by the wedge_sonar_line.

Considering the worst case error, we can set $\alpha_1 = \alpha_2 = \theta$. Assuming that the error in θ is zero, then the error in the calculated points is:

$$\Lambda_{x_i} = \left(\frac{\partial x_i}{\partial r}\right)^2 \Lambda_r$$

$$\Lambda_{z_i} = \left(\frac{\partial z_i}{\partial r}\right)^2 \Lambda_r$$

which results in:

$$\Lambda_{x_1} = \cos^2\theta\, \Lambda_r, \quad \Lambda_{z_1} = \sin^2\theta\, \Lambda_r$$

$$\Lambda_{x_2} = \cos^2\theta\,\Lambda_r, \quad \Lambda_{z_2} = \sin^2\theta\,\Lambda_r$$

Finally, the error function for $ILSS2$ is:

$$error(ILSS2) = \{\Lambda_{x_1}, \Lambda_{z_1}, \Lambda_{x_2}, \Lambda_{z_2}\}$$

As an example, if $\Lambda_r = 4.0mm^2$, and $\theta = 11^o$ (approximately correct for the Polaroid sensor), we get:

$$error(ILSS2) = \{3.85mm^2, 0.15mm^2, 3.85mm^2, 0.15mm^2\}$$

This example illustrates the importance and usefulness of the ILSS library since all these analyses can be performed once and put in the library for reuse and the user does not have to go through these details again. For example, if a different sonar sensor is used, then the same error analysis can be used by supplying the sensor's error variance. In addition, given that the error range has been determined, redundancy can be added using different sensor pairs to sense the same wall and a monitor can be added to detect error discrepancies.

5.3 Experimental Results

We do not have a very good model of our camera, and therefore actual experiments were required to compare the pose error for the two proposed techniques. The two instrumented logical sensors were used with the LABMATE to find the location of walls using real data. The goal of the experiment was to use the framework to obtain measures to help choose between a vision based wall pose technique and a sonar based wall pose estimator.

First, we calibrated the range of our visual target (a horizontal line at a known height, Y_0 with vertical stripes regularly spaced 34.2mm apart) with its y-location in the image. This was done by aligning the z-axis of the mobile robot camera to be normal to the wall; the mobile robot was then backed away from the wall a known distance and the image row number of the horizontal target line recorded. Figure 12 shows the results of this step. (Note that we digitized a 128x128 image; greater resolution would produce more accurate results.)

Once the target range calibration was done, the robot was placed in eight different poses with respect to the wall and the visual target acquired. Each image was constrained to have at least two vertical stripes and neither of them could be centered on the middle column of the image. The test images are shown in Figure 13.

Sonar data was also taken at each pose. The actual pose of the mobile robot with respect to the wall was independently measured by hand. Table 1 gives the hand measured, sonar and image calculated results.

The error values of the sonar and vision results with respect to the handmeasured data are plotted in Figures 14 and 15.

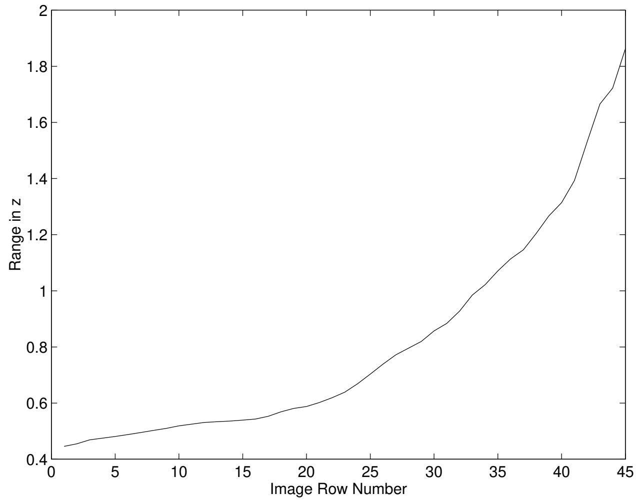

Figure 12: Row vs. range

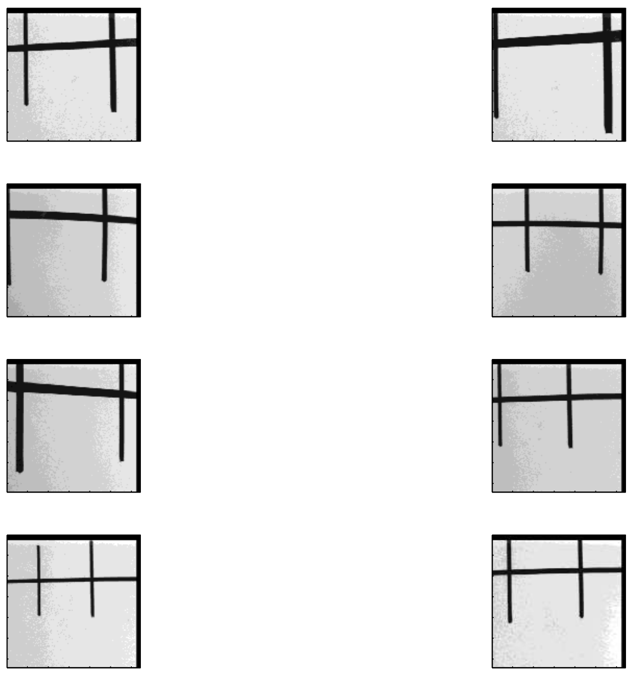

Figure 13: Visual target test images

Test No.	*Measured ρ*	*Measured θ*	*Sonar ρ*	*Sonar θ*	*Vision ρ*	*Vision θ*
1	919	-21	915.6	-20.6	888	-29.66
2	706	-27	715.4	-22.7	667	-35.51
3	930	20	924.0	23.2	783	23.99
4	1,242	0	1,226.3	4.6	1,128	10.27
5	764	32	778.5	46.1	593	43.62
6	1,164	-11	1,164.9	-13.7	1,084	-13.33
7	1,283	6	1,277.4	3.7	979	-6.53
8	1,319	-10	1,300.8	-9.8	1,084	-13.33

Table 1: Pose results from measured data, sonar, and vision techniques.

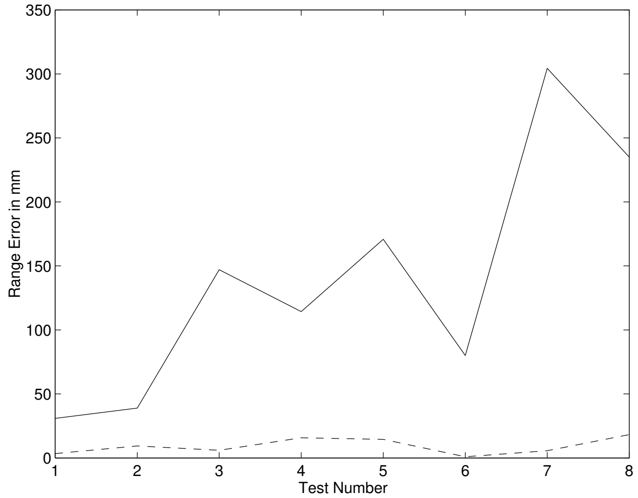

Figure 14: Error in ρ for sonar (dashed line) and vision

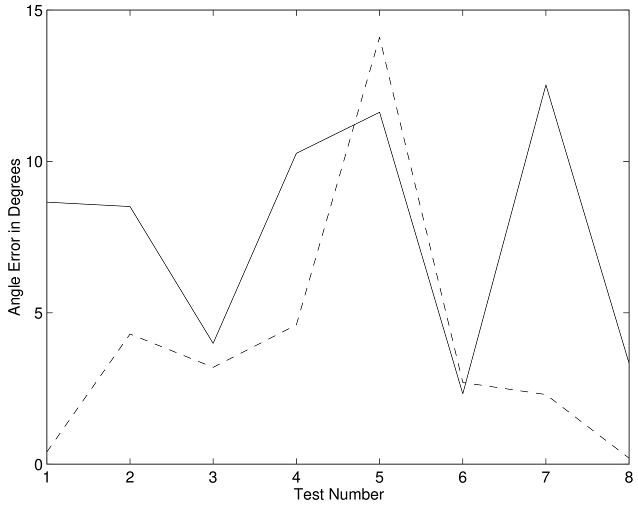

Figure 15: Error in θ for sonar (dashed line) and vision

These results allow the user to decide whether to use one technique or the other given the global context. For example, our application was a tennis ball pickup competition in which we were using vision to track tennis balls anyway, and we needed to locate a delivery location along the wall; if we can get by with pose error of less than $0.3m$ range and 15^o angle, then ILSS1 will suffice. If less error were required, then a costly sonar system with hardware and software would need to be added to the robot, or else the use of higher resolution imagery could be explored. However, decisions made with respect to all these considerations would now be defensible and well documented. (For another detailed example comparing two alternative sonar sensor techniques to obtain wall pose, see [19].)

Note that, to keep things simple, we did not consider the error in the sonar location and orientation. However, these errors can be incorporated into the model in the same manner.

6 Conclusions

In this paper we presented a theoretical framework for sensor modeling and design based on defining the performance semantics of the system. We introduced the notion of *instrumented sensor systems*, which is a modeling and design methodology that facilitates interactive, on-line monitoring for different components of the sensor system. It also provides debugging tools and analysis measures for the sensor system. The instrumented sensor approach can be viewed as an abstract sensing machine which defines the semantics of sensor systems. This provides a strong computational and operational engine that can be used to define and propagate several quantitative measures to evaluate and compare design alternatives. The implementation of this framework within the CWave system was described and examples were presented.

Currently, we are working on building an ILSS library with several design tools which will assist in rapid prototyping of sensor systems and will provide an invaluable design tools for monitoring, analyzing and debugging robotic sensor systems.

Acknowledgment

We would like to thank Professor Robert Kessler and Christian Mueller for providing the CWave program that we used to implement the instrumented sensor library, Professor Gary Lindstrom for his helpful discussions of program semantics, and Kevin Linen of North Carolina A & T for help with the experiments.

References

[1] ASHCROFT, E. A. R for semantics. *ACM Transactions on Programming Languages and Systems 4*, 2 (1982), pp. 283–295.

[2] BRIGGS, A., AND DONALD, B. Automatic sensor configuration for task-directed planning. In *IEEE Int. Conf. Robotics and Automation* (May 1994), pp. 1345–1350.

[3] BROOKS, R. R., AND IYENGAR, S. Averaging algorithm for multi-dimensional redundant sensor arrays: resolving sensor inconsistencies. Tech. rep., Louisiana State University, 1993.

[4] DEKHIL, M., AND HENDERSON, T. C. Instrumented sensor systems. In *IEEE International Conference on Multisensor Fusion and Integration (MFI 96), Washington D.C.* (December 1996), pp. 193–200.

[5] DEKHIL, M., AND HENDERSON, T. C. Optimal wall pose determination in a shared-memory multi-tasking control architecture. In *IEEE International Conference on Multisensor Fusion and Integration (MFI 96), Washington D.C.* (December 1996), pp. 736–741.

[6] DEKHIL, M., AND SOBH, T. M. Embedded tolerance analysis for sonar sensors. In *Invited paper to the special session of the 1997 Measurement Science Conference,* Measuring Sensed Data for Robotics and Automation*, Pasadena, California* (January 1997).

[7] DONALD, B. R. On information invariants in robotics. *Artificial Intelligence*, 72 (1995), pp. 217–304.

[8] DURRANT-WHYTE, H. F. *Integration, coordination and control of multisensor robot systems.* Kluwer Academic Publishers, 1988.

[9] FAUGERAS, O. *Three-dimensional computer vision - a geometric viewpoint.* The MIT Press, 1993.

[10] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design patterns: elements of reusable object-oriented software.* Addison Wesley, 1995.

[11] GIRAUD, C., AND JOUVENCEL, B. Sensor selection in a fusion process: a fuzzy approach. In *Proceedings of the IEEE International Conference on Multisensor Fusion and Integration* (Las Vegas, NV, 1994), R. Luo, Ed., IEEE, pp. 599–606.

[12] GORDON, M. J. C. *Denotational description of programming languages.* Springer-Verlag, 1979.

[13] GROEN, F. C. A., ANTONISSEN, P. P. J., AND WELLER, G. A. Model based robot vision. In *IEEE Instrumentation and Measurment Technology Conference* (1993), pp. 584–588.

[14] HAGER, G., AND MINTZ, M. Computational methods for task-directed sensor data fusion and sensor planning. *Int. J. Robotics Research 10*, 4 (August 1991), pp. 285–313.

[15] HAGER, G. D., AND MINTZ, M. Task-directed multisensor fusion. In *IEEE Int. Conf. Robotics and Automation* (1989).

[16] HENDERSON, T. C., BRUDERLIN, B., DEKHIL, M., SCHENKAT, L., AND VEIGEL, L. Sonar sensing strategies. In *IEEE Int. Conf. Robotics and Automation* (April 1996), pp. 341–346.

[17] HENDERSON, T. C., AND DEKHIL, M. Visual target based wall pose estimation. Tech. Rep. UUCS-97-010, University of Utah, Department of Computer Science, July 1997.

[18] HENDERSON, T. C., DEKHIL, M., BRUDERLIN, B., SCHENKAT, L., AND VEIGEL, L. Flat surface recovery from sonar data. In *DARPA Image Understanding Workshop* (February 1996), pp. 995–1000.

[19] HENDERSON, T. C., DEKHIL, M., BRUDERLIN, B., SCHENKAT, L., AND VEIGEL, L. Wall reconstruction using sonar sensors. *To appear in the IEEE International Journal of Robotics Research* (1997).

[20] HENDERSON, T. C., HANSEN, C., AND BHANU, B. The specification of distributed sensing and control. *Journal of Robotic Systems* (Mar. 1985), pp. 387–396.

[21] HENDERSON, T. C., AND SHILCRAT, E. Logical sensor systems. *Journal of Robotic Systems* (Mar. 1984), pp. 169–193.

[22] HOLMAN, J. P., AND W. J. GAJDA, J. *Experimental methods for engineers.* McGraw-Hill, 1978.

[23] HU, H., BRADY, J. M., DU, F., AND PROBERT, P. J. Distributed real-time control of a mobile robot. *Journal of Intelligent Automation and Software Computing* (1995).

[24] IYENGAR, S. S., AND PRASAD, L. A general computational framework for distributed sensing and falt-tolerant sensor integration. *IEEE Trans. Systems Man and Cybernetics* (May 1994).

[25] JOSHI, R., AND SANDERSON, A. C. Model-based multisensor data fusion: a minimal representation approach. In *IEEE Int. Conf. Robotics and Automation* (May 1994), pp. 477–484.

[26] KAPUR, R., WILLIAMS, T. W., AND MILLER, E. F. System testing and reliability techniques for avoiding failure. *IEEE Computer* (November 1996), pp.28–30.

[27] KIM, K. H., AND SUBBARAMAN, C. Fault-tolerant real-time objects. *Communications of the ACM 40*, 1 (January 1997), pp.75–82.

[28] NADIG, D., IYENGAR, S. S., AND JAYASIMHA, D. N. New architecture for distributed sensor integration. In *IEEE SOUTHEASTCON Proceedings* (1993).

[29] PRASAD, L., IYENGAR, S. S., KASHYAP, R. L., AND MADAN, R. N. Functional characterization of fault tolerant integration in distributed sensor networks. *IEEE Trans. Systems Man and Cybernetics* (September 1991), pp. 1082–1087.

[30] PRASAD, L., IYENGAR, S. S., RAO, R. L., AND KASHYAP, R. L. Fault-tolerence sensor integration using multiresolution decomposition. *The American Physical Society* (April 1994), pp. 3452–3461.

[31] PROFETA, J. A. Safety-critical systems built with COTS. *IEEE Computer* (November 1996), pp.54–60.

[32] SCHENKAT, L., VEIGEL, L., AND HENDERSON, T. C. Egor: Design, development, implementation – an entry in the 1994 AAAI robot competition. Tech. Rep. UUCS-94-034, University of Utah, Dec. 1994.

[33] SCHNEIDER, S. A., CHEN, V., AND PARDO, G. Controlshell: A real-time software framework. In *AIAA Conference on Intelligent Robots in Field, Factory, Service and Space* (1994).

[34] SIMON, D., ESPIAU, B., CASTILLO, E., AND KAPELLOS, K. Computer-aided design of a generic robot controller handling reactivity and real-time issues. *IEEE Transactions on Control Systems Technology 4*, 1 (1993).

[35] STEWART, D. B., AND KHOSLA, P. K. Mechanisms for detecting and handling timing errors. *Communications of the ACM 40*, 1 (January 1997), pp.87–93.

[36] WELLER, G. A., GROEN, F. C. A., AND HERTZBERGER, L. O. A sensor processing model incorporating error detection and recoverry. In *Traditional and non-traditional robotic sensors.* Edited by T. C. Henderson. (1990), Springer-Verlag, pp. 351–363.