0 2 / 3 4 /

INTELLIGENT AGENT ARCHITECTURES:
Reactive Planning Testbed

Final Technical Report

December 19, 1993

Government Contract No. NAS2-13229

Prepared for:

National Aeronautics & Space Administration
Ames Research Center
COTR, Code FIA, M/S: 244-7
Moffett Field, CA 94035-1000

Prepared By:

Stanley J. Rosenschein
Philip Kahn

UNCLASSIFIED

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

List of Figures

1 EXECUTIVE SUMMARY

1.1 Nature of the Problem

An Integrated Agent Architecture (IAA) is a framework or paradigm for constructing intelligent agents. Intelligent agents are collections of sensors, computers, and effectors that interact with their environments in real time in goal-directed ways. Because of the complexity involved in designing intelligent agents, it has been found useful to approach the construction of agents with some organizing principle, theory, or paradigm that gives shape to the agent's components and structures their relationships.

Given the wide variety of approaches being taken in the field, the question naturally arises: Is there a way to compare and evaluate these approaches? The purpose of the present work is to develop common benchmark tasks and evaluation metrics to which intelligent agents, including complex robotic agents, constructed using various architectural approaches can be subjected.

1.2 Program Benefits

Benefits derived from a shared set of benchmarks and metrics for Integrated Agent Architectures include:

1. Establishes the value of agent architectures
   - *Benchmark tasks and evaluation metrics provide some criteria for assessing agent methodologies*
2. Guides research in agent architectures
   - *Benchmarks allow researchers to formulate hypotheses about their agents in experimentally verifiable ways, and then carry out specific tests to validate these hypotheses*
3. Guides end-users in procuring agent architectures
   - *Application developers can choose agent methodologies that perform well for selected benchmarks relevant to their application domain*
4. Advances state of the art in agent architecture evaluation
   - *Little work has been done to date in the definition of benchmark tasks for intelligent agents in physical domains. Evaluation metrics that encompass physical agents and operational factors are needed.*
5. Technology transfer
   - *The developed IAA Testbed and evaluation metrics, combined with Teleos' background, experience, and facilities, offer special opportunities for technology transfer to other research, development and ARPA programs.*

1.3 Program Goals

This program has the following goals:

1. Define a canonical testbed domain in which agent methodologies may be applied. This domain should have the following properties:
   a. Relationships to established planning problems
   b. Means-end complexity (state-space description)
   c. Problems of sufficient complexity to benefit from concurrent agents, opportunism, and richness of goals
   d. Domain realism and utility
   e. Asynchronous events occur which require agent attention
   f. Human awareness and interactivity
   g. Uncertainty, dynamism and errors (e.g., perception errors)
   h. A capability to objectively measure performance (i.e., scoreable tasks)
   i. Controllable degrees of difficulty
   j. Implementation feasibility.
2. Develop a set of agent evaluation metrics and scoring methodologies that can be used to evaluate agent performance.
3. Define a Testbed Application Programmer's Interface to be used by agents to operate within the domain.
4. Physically implement the Testbed API including effectors, visual perception and proprioception.
5. Implement a sample agent using the Testbed API to perform benchmark tasks.
6. Evaluate the performance of the sample agent using developed evaluation metrics and scoring methods.

1.4 Program Accomplishments

1. Successfully achieved all program goals
2. A major contribution of the work reported here has been to address the difficulties inherent to physical and robotic domains, and to present a realistic evaluation methodology for high-level intelligent agents in robotic domains
3. Sponsored the First Workshop on Benchmarks and Metrics for Integrated Agent Architectures to solicit opinions and advice concerning proper evaluation methodology from leading researchers in the field
4. Teleos played a leadership role in organizing a National Virtual Laboratory (NVL) for perception and control to foster collaboration among researchers in government, industry and academia.
5. Delivered a Final Technical Report describing this work.

1.5 Conclusions and Recommendations

Evaluation of integrated agent architectures operating in realistic domains is important for understanding and guiding research and application of these agents. The work presented in this

report provides a better understanding of how such evaluations may be made and how future work may be guided.

Further work is needed to better develop the Testbed API to expand the range of applicable problem domains and derive additional agent evaluation metrics. Continuation of National Virtual Laboratory efforts should expand the application and development of the understanding and capabilities that were developed under this effort.

1.6 Structure of this Report

Section 2 discusses Integrated Agent Architectures and related work.

Section 3 describes the testbed agent evaluation domain.

Section 4 discusses methods for evaluating the performance of agents operating within the testbed domain.

Section 5 describes the Testbed Application Programmer's Interface (API) that is available to agents.

Section 6 describes the physical implementation of the Testbed API using robotic perception and manipulation systems.

Section 7 describes a sample agent implementation, and it provides an analysis of its performance using computed agent evaluation metrics and scoring methods.

Section 8 provides some closing remarks.

2 INTRODUCTION

2.1 Integrated Agent Architectures

This section defines the notion of an Integrated Agent Architecture, motivates the need for a formal evaluation process, and discusses general issues that arise in the evaluation of agent architectures.

2.1.1 What Integrated Agent Architectures Are

An Integrated Agent Architecture (IAA) is a framework or paradigm for constructing intelligent agents. To understand such architectures, we must first consider the nature of intelligent agents.

Intelligent agents are collections of sensors, computers, and effectors that interact with their environments in real time in goal-directed ways. As such, an intelligent agent can be regarded as a type of closed-loop feedback control system. As with all such control systems, the agent's sensors detect conditions in the environment, the computational system maps these sensory data into effector actions, which in turn influence the state of the environment and cause desired events to ensue. The main difference between intelligent agents and conventional control systems lies in their "intelligence" — an admittedly difficult attribute to define. For our purposes, "intelligence" is best viewed as a measure of the diversity of goals the agent can pursue, the breadth of situations in which these goals can be successfully achieved, and the complexity of means-ends relationships involved in pursuing those goals. In the absence of operational detail, these concepts remain vague, but researchers in the field share broad intuitions about them, and indeed much of the motivation for the present work lies precisely in augmenting those intuitions with solid operational criteria that can be more objectively studied.

Because of the complexity involved in designing intelligent agents, it has been found useful to approach the construction of agents with some organizing principle, theory, or paradigm that gives shape to the agent's components and structures their relationships. As with many engineering problems, the design of complex control systems involves the identification of major subsystems and modules, characterization of their inputs and outputs, and the definition of how the overall behavior of the system results from interactions of these components. The success or failure of engineering efforts often depends crucially on the workability of these architectural choices. This is as true for artificial intelligent agents as for any other complex artifact: having a solid system architecture is an essential prerequisite for success.

2.1.2 What Integrated Agent Architectures Are Good For

In principle, one could define new architectures, with entirely different components and relationships, for each individual application. However, where classes of application share some common structure, it is clearly desirable to take advantage of the shared attributes by identifying a generic architectural schema that can simply be filled in to yield solution instances. In this way, general properties of the schema can be characterized and understood in advance and the work of the developers of specific applications made that much easier. It is the working hypothesis of researchers in Integrated Agent Architectures that such generic schemas do indeed exist for intelligent agents, and with this hypothesis as their starting point, these researchers have set as

their goal the development of generic structures that characterize broad classes of intelligent reactive systems.

2.1.3 Some Examples

The field of Integrated Agent Architecture contains a large number of paradigms of very different sorts. At one extreme, there are frameworks that are primarily conceptual in nature — ways of thinking about agent construction. At the other extreme are architectures that include a strong commitment to particular software structures and programming tools. A few examples should suffice to illustrate the range of work that has been included under the heading of integrated agent architectures.

One key dimension of variation among architectural approaches has to do with the role of run-time symbolic reasoning. This is an area of controversy in the field, with some researchers absolutely committed to symbolic reasoning, others equally opposed, and still others neutral or willing to use symbolic reasoning either off-line or as part of a hybrid system. The tradition of reasoning about action is very strong in Artificial Intelligence (AI) and lies at the basis of models of integrated agents going back to classical work on the General Problem Solver (GPS) [48] and the STRIPS planning system used to control Shakey the robot [28]. Shakey's action-generation system is an interesting case study as it includes, in effect, a mechanism for generating and executing reactive plans and thus foreshadows much recent work in reactive planning.

The influence of the classical planning model, with a new emphasis on reactivity, can be seen most clearly in the work of several groups of Integrated Agent researchers. Their work includes Georgeff and Lansky's Procedural Reasoning Systems (PRS) [31], Laird's SOAR architecture [41], work by Barbara Hayes-Roth [33] on high-level, real-time control, and work by Bratman, Israel and Pollack on architectures for rational agents [10]. Each of these researchers proposes architectural components structures based on models (different in each case) that can be regarded as models of a reasoning process.

Another broad category of architecture is based on direct representations of situation-action mappings, formalisms for expressing these mappings, and mechanisms for applying them at run-time. One subcategory includes circuit models of situation-action maps. These have been investigated by Nilsson [49] in his work on action nets and by Rosenschein and Kaebling in their work on situated automata [58]. Related work includes Schoppers investigations of Universal Plans [62] and work by Agre and Chapman on Pengi [1]. Drummond's work on situated control rules and related mechanisms [23] is a blend of classical planning approaches and reactive control, in which non-determinism is gradually reduced as constraints are brought to bear, decisions made, and actions taken based on a representation that is semantically close to that used in classical planning.

2.2 Benchmarks and Metrics for Integrated Agent Architectures

Given the wide variety of approaches being taken in the field, the question naturally arises: Is there a way to compare and evaluate these approaches? The purpose of the present work is pursue one particular evaluation methodology: the use of common benchmark tasks and evaluation metrics to which intelligent agents constructed using various architectural approaches can be subjected.

The design of meaningful benchmarks carries several difficult challenges. It is important to distill the essential features that define intelligent-agent systems as a class. These features must be characteristic of application domains, on the one hand, but must be simple and broadly representative. Finally, the tasks must be complex enough to stress real agent systems but not so complex that they are beyond the state of art.

In the work we have done at Teleos, we have chosen to focus on a particular class of benchmarks, namely those involving physical tasks performed by robotic agents. This was done for several reasons: (1) Robotic agents are an active area of research and much in need of performance measurement. (2) This focus complements that of our research collaborators, and others in the field, who have pursued work on evaluation using synthetic agents. (3) Our background, experience, and facilities offer special opportunities for making a meaningful contribution in this area, with the possibility of transfer to other ARPA programs.

Within the broad area of robotic agents, we have focused on hand-eye systems for the following reasons: (1) hand-eye tasks are paradigmatic of tasks involving close coordination of perception and action; (2) significant events happen on the order of seconds or less, unlike robot navigation which tends to involve many tens of seconds or minutes between significant events; (3) hand-eye tasks lend themselves to greater task diversity, and this allows measurement of the performance of planners and reactive systems on tasks involving chains of dependent conditions. This section discusses some of the general benefits to be derived from a shared set of benchmarks and metrics for Integrated Agent Architectures.

2.2.1 Establishing the Value of Agent Architectures

Agent Architectures are conceptual inventions. They are typically proposed because they appeal in their form to their inventors or because they capture some pre-existing intuition about how perception, reasoning, action-generation, or other such process should be conceptualized, modeled, or implemented. As such, these architectures, like other abstractions, can have a purely aesthetic or analytical appeal. On the other hand, it does not follow that a given architectural model actually yields superior performance in agents, and it is often unclear how one would measure that performance. One reason for defining a set of benchmark tasks and evaluation metrics is precisely to provide some objective criterion or yardstick for which the performance of agent methodologies can be measured.

This approach to the evaluation of complex systems has been applied successfully in numerous other fields, most notably in the evaluation of computer processor units and software compilers. In both these fields, although the utility of specific benchmarks are often hotly debated, the existence of benchmarks provides a common language for carrying out the debate and are widely used.

2.2.2 Guiding Research in Agent Architectures

Another direct benefit of benchmarks and metrics is in guiding the activities of researchers themselves. A researcher who proposes a specific architecture does so, in many instances, because he believes it will produce an effective solution for a class of problems. The existence of benchmarks allows researchers to formulate clear, experimentally verifiable hypotheses about their agents. They may then carry out specific tests to validate these hypotheses. Failure of the

agent to perform adequately on benchmark tasks focuses attention on architectural choices responsible for the inadequacies. Architectural flaws are more readily detected and understood through the use of these standard benchmarks than by ad-hoc experimentation methodologies.

2.2.3 Guiding End-Users in Procuring Agent Architectures

There is also a benefit to be gained outside the research process itself. Intelligent agents are of high interest because of their potential practical value. Just as control systems are to be found in a wide variety of applications and embedded systems, there is enormous potential for incorporating intelligent-agent control processes into a wide range of future applications. Sample applications that could benefit from intelligent agents include network database browsers, factory control systems, smart appliances, and others.

Designers of such systems are faced with a wide array of choices regarding conceptual approaches. The availability of benchmark tasks and evaluation metrics will allow these application developers to select benchmarks that bear attributes characteristic of their application domain and to evaluate objectively how an agent constructed under a particular architectural methodology fared on those benchmarks. Having such an objective evaluation technique, should be of great value in advancing practical applications.

2.2.4 The State of the Art in Agent Architecture Evaluation

Most of the work done to date on the evaluation of agent architectures has been done in the context of simulated software worlds. Notable in this category are two systems named TileWorld, one developed by Martha Pollack of SRI and more recently the University of Pittsburgh, and the other by Mark Drummond of NASA Ames Research Center. In each of these systems, a simulated agent is given tasks to perform in a simulated environment which it can sense and act upon. Unpredictable, exogenous events can happen, and measurements can be taken of the agent's performance. These measurements can then be analyzed and conclusions drawn about agent performance. The Phoenix system from the University of Massachusetts explored similar agents operating in a real-time fire-fighting planning domain [18].

Little has been done to date in the definition of benchmark tasks for intelligent agents in physical domains. Part of the reason for this has to do with two central difficulties. The first is in the provision of suitably high level perception and action primitives. Most research in intelligent agent architectures is at a level of abstraction far above that supported in conventional robotic environments. The difficulties in bridging between layers of abstraction is a major impediment to systematic evaluation of intelligent agent architectures in robotic domains. The second difficulty lies in controlling the numerous confounding factors that physical experiments are prone to. A major contribution of the work reported here has been to address these difficulties and to present a realistic evaluation methodology for high-level intelligent agents in robotic domains.

2.3 Shared Tasks and Methods for the Agent Architecture Community

Apart from any advantages to be derived by individual researchers and users through the application of benchmarks and metrics, there is a social phenomenon at work in the very process by which a community forms consensus about benchmarks and metrics. One of the aims of this project has been to stimulate that social process. This has been pursued both informally through

discussions with colleagues about the aims and progress of our project, as well as through the formal mechanisms of workshops and participation in the early stages of the development of a unique experiment, the National Virtual Laboratory. These activities are described briefly below.

2.3.1 The Benchmarks and Metric Workshop

Early in the course of this project, Teleos, under the joint sponsorship of NASA and ARPA, held a workshop on Benchmarks and Metrics at the NASA Ames Research Center on June 25, 1990. The objective of this workshop was to solicit opinions from leading researchers in the intelligent-agent research community and initiate a process of dialogue regarding the formal evaluation of agent architectures. The workshop was attended by several dozen researchers and was successful in eliciting a range of opinion, many of which are presented in the document reproduced as an appendix to this report. The workshop also had a role in bringing the question of evaluation into focus and influencing the production of numerous papers and articles on aspects of agent evaluation. The output of the Workshop also directly affected subsequent work on the Teleos testbed project.

2.3.2 National Virtual Laboratory

In February 1993, a meeting was held at ARPA in which about two dozen real-time planning and control researchers participated. The aim of that meeting was to help ARPA formulate strategies for creating, evaluating, and applying autonomous-systems technology over the next few years. One outcome of that meeting was a strong consensus among the participants that unique opportunities exist to accelerate technical progress dramatically through innovative approaches to the sharing of resources, facilities, software modules, and technical results.

It was recommended that ARPA create an infrastructure for such sharing through an experimental "national virtual laboratory." The laboratory would consist of numerous physically distributed sites connected by video, voice, and data links offering high-quality, real-time interchange and remote operation of facilities. Beyond advanced communications media, the laboratory would provide mechanisms (involving computers and humans) to actively stimulate sharing in service of technology creation and diffusion. Examples of such mechanisms include: active search for technology and application linkages (simple lack of information is often the major obstacle to sharing); focused, small-dose funding to rapidly package and document software modules for re-use by others; and facilitation of distributed rapid-prototyping efforts that draw heavily on existing components and expertise from multiple sites. Attention would be paid to maximizing the impact for minimum incremental investment. The net effect would be to multiply the effectiveness of the research community by enabling widespread, real-time access to pockets of local expertise and resources and linkages to application problems.

It was further suggested that the autonomous-systems research community take the lead in pioneering this concept, and it was also suggested that this group might design a community mini-experiment, perhaps centering around rapid-prototyping of an interactive robotic demonstration system.

The idea of participating in such an experiment, and in the virtual laboratory as a whole, was very exciting to the group. If successful, it could help catalyze new organizational paradigms and supporting technology for accelerating and enhancing the research enterprise on a national scale.

Since that meeting in February, Teleos has been in the forefront of efforts to carry forward the idea of a National Virtual Lab in real-time perception and control. Through continuing dialogue with researchers, government agencies, and potential application developers, some initial experiments began to be defined, including an experiment between Brown University and the University of Pennsylvania. This experiment is now being expanded to include Stanford University and Teleos, and it involves putting components of the Teleos agent-architecture testbed on-line. One outcome of these experiments is the beginning of the definition of network protocols for controlling real-time systems remotely and common high-level abstractions for robot control. The continuation of this collaborative process, along with access to common benchmark problems, is very much in the spirit of the Teleos testbed project and potentially one of its most tangible positive results.

3 INTEGRATED INTELLIGENT AGENTS AND THE TESTBED DOMAIN

This section describes the testbed problem domain developed to evaluate IAA performance.

3.1 The Need to Define a Testbed Benchmark Task

In order to perform benchmarks of Integrated Intelligent Agent (IAA) performance, computational performance metrics must be defined and measured. The quantification of performance metrics requires a concrete testbed problem domain in which the metrics vary and describe aspects of agent performance. The remainder of this section describes the testbed domain selection criteria, the chosen domain, and specific agent tasks within the context of the testbed domain.

3.2 Desired Testbed Properties

It is important that the testbed domain possess properties that demonstrate and challenge operational properties of IAA. In considering possible testbed domains, we derived the following set of desirable testbed properties:

- Relationships to established planning problems
- Means-end complexity (state-space description)
- Problems of sufficient complexity to benefit from concurrent agents, opportunism, and richness of goals
- Domain realism and utility
- Asynchronous events occur which require agent attention
- Human awareness and interactivity
- Uncertainty, dynamism and errors (e.g., perception errors)
- A capability to objectively measure performance (i.e., scoreable tasks)
- Controllable degrees of difficulty
- Implementation feasibility.

When developing the testbed problem domain, the need for implementational feasibility is in conflict with the other desired domain properties, which tend to increase complexity and realism. The challenge in developing the testbed was thus determining a bound and realistic set of implementationally feasible primitives that can fit into a domain that well meets the other desired properties found in a good IAA testbed domain.

3.3 Relationship between the Intelligent Agent and the Testbed

The testbed specifies interfaces to perception and action which an agent may use to interact with the testbed problem domain in order to achieve its goals. As shown in Figure 1, the physical *testbed equipment* interacts with the environment and testbed domain according to its physical and operational properties. Examples of such equipment could include video sensors and their processors, robot arms and manipulators, force sensing, and acoustical input.

The *testbed Application Programmer's Interface (testbed API)* provides a common software syntax and operational semantics for the physical testbed equipment used by the intelligent agent to perceive environmental state and effect action in the environment. The testbed API provides support for relatively isolated sensorimotor function (e.g., "find the depth at this position," "move the end effector to *(x,y,z)*"). Optionally, the testbed may also include more integrated *programmed behaviors* which provide a more abstract level of agent interaction. For example, the testbed API may provide support for measuring depths, motion, and tracking; a programmed behavior may integrate these capabilities to "find a person's head." As will be discussed in more detail later, the use of such programmed behaviors can both simplify and complicate how an agent may effectively operate.

The testbed and its API are not specific to any single intelligent agent or methodology. Indeed, their entire purpose is to provide a standardized environment in which agents may operate and be objectively evaluated. To test an intelligent agent in the testbed domain requires that an agent ground itself in the testbed equipment using the testbed API and programmed behaviors. For example, a CommonLISP-based testbed API only requires that the testbed agent be capable of invoking the appropriate CommonLISP functions. As described in later sections, a CommonLISP-based API was developed for the testbed since many intelligent agent systems used today are implemented in CommonLISP. The intent of the testbed and its API is to provide relatively straightforward plug-and-play capability so that a wide range of intelligent agent techniques and systems may be objectively evaluated.
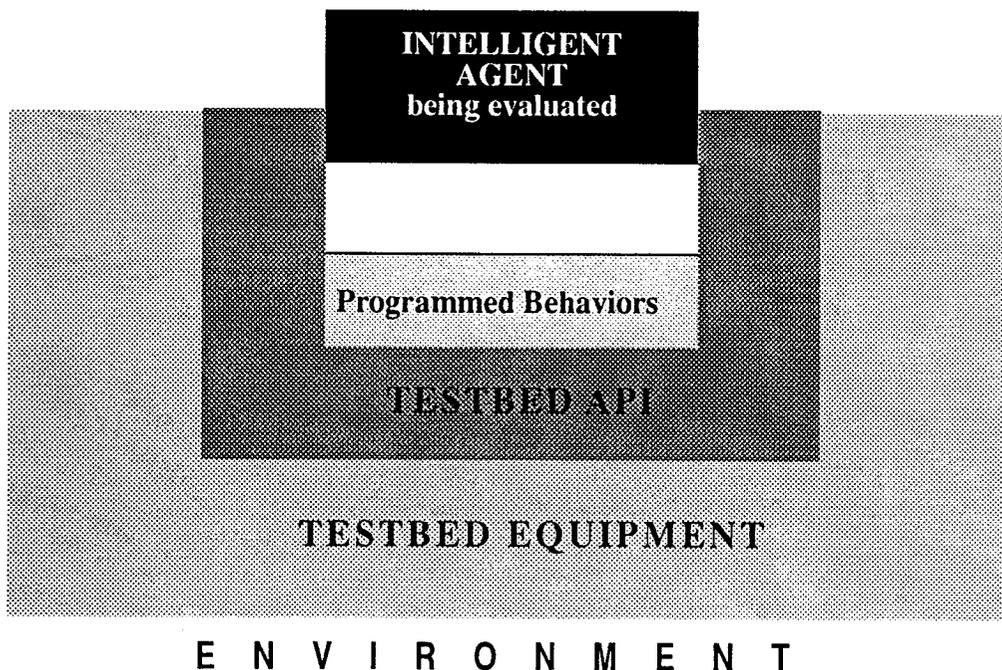
Figure 1: Relationship between the Intelligent Agent and the Testbed

3.4 Testbed Problem Domain: Package Counter Service Robot (PC-Serbot)

A Package Counter Service Robot (PC-Serbot) domain was chosen for the testbed since it exemplifies many of the desired characteristics described in Section 3.2. Generically, package handling domains require the identification of packages, manipulation of packages whose properties may be incompletely modelled, interfacing with the producers and consumers of packages (e.g., humans), organizing its workspace, scheduling, graceful handling of uncertainty and errors, and other issues. Common examples of package counter domains include UPS and Post Office service counters ("I want to ship/receive a package"), fast food counter service ("I want to buy a cookie"), and warehousing ("I need a box of #1/4-20 bolts").

The package handling domain has a direct relationship to classical AI planning problems which have been much explored in "blocks world stacking problems." Hence, these domains provide a well understood planning problem while still addressing a general class of real world applications of significance and utility. The interpretive value of the testbed is significantly enhanced by its ability to judge performance within both a classical planning problem domain (e.g., blocks world) as well as problem areas addressed by IAA for which no canonical problem set has yet been defined.

Figure 2 shows the PC-Serbot testbed domain and apparatus that was developed and implemented under this project. Human customers may walk up to a *service counter* in order to pick up or drop off a package. A package dropped off may subsequently be requested for pick up by a later customer.

In this benchmark domain, the intelligent agent has two main tasks or transactions that it must execute:

- find a desired package in a stocking area and give it to a customer (i.e., a *pick-up*), or,

- get a package from the customer, process it, and place it in a stocking area (i.e., a *drop-off*). The metrics used to evaluate an intelligent agent and its performance will quantify how well the agent performs on aspects of these two primary tasks (i.e., transactions).

3.5 Example Package Counter Domain Actions

This subsection is intended to motivate aspects of the package counter domain. These characterizations provide example agent actions in the general package domain, though agents can choose to chunk the problem and its solution in different ways. A specific description of the benchmark tasks is given in Section 3.6, and an agent is evaluated on their performance in solving these benchmark tasks.

Figure 3 shows six example elements in the package counter robot (PC-Serbot) domain:
- Detection of Service Need
    - Detect whether a customer is at the counter requiring service
- Identification of Task (Transaction) Type
    - Determine whether the customer wants to pick up or drop off a package
- Execution of Activity
    - Receive package from customer and store in stocking area
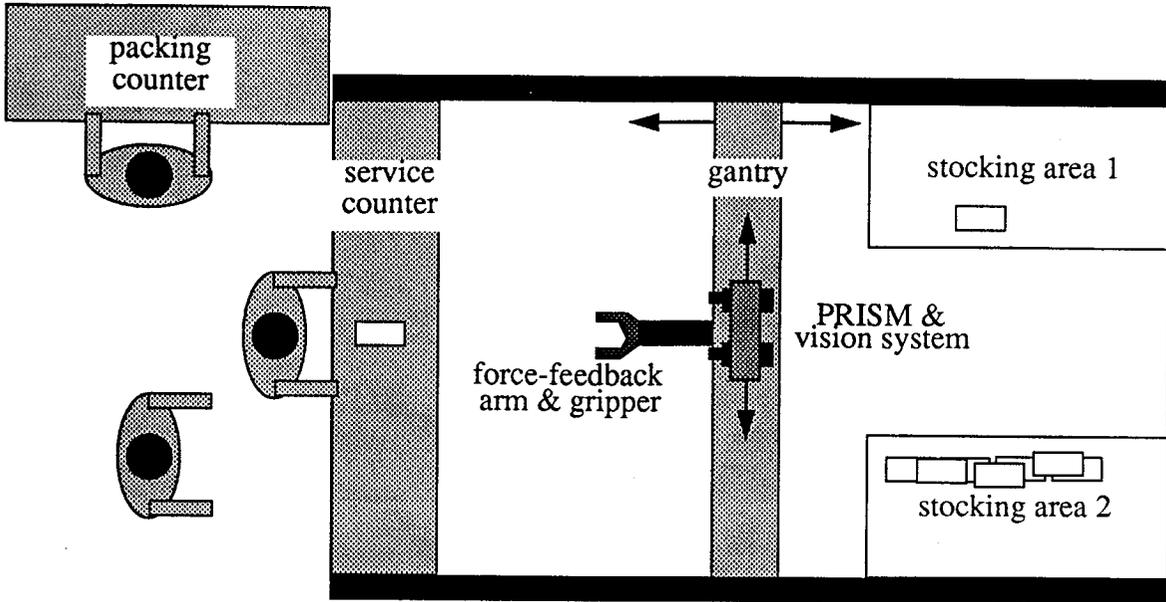
Figure 2: PC-Serbot Scenario Apparatus

- Find and retrieve package from stocking area
- Clarification Act

    At times, the agent may benefit from engaging the customer in a dialog in order to acquire more information about its task or the packages it is handling. For example, if two packages meet the agent's retrieval criteria, the agent may choose to acquire focused package information from the customer in order to perform the final selection. Or, if a package retrieval search is taking a long time or others are waiting, the agent may elect to engage in customer dialog in order to better focus its efforts (e.g., "did you get the notice within the last week?" may help to more efficiently order package search).
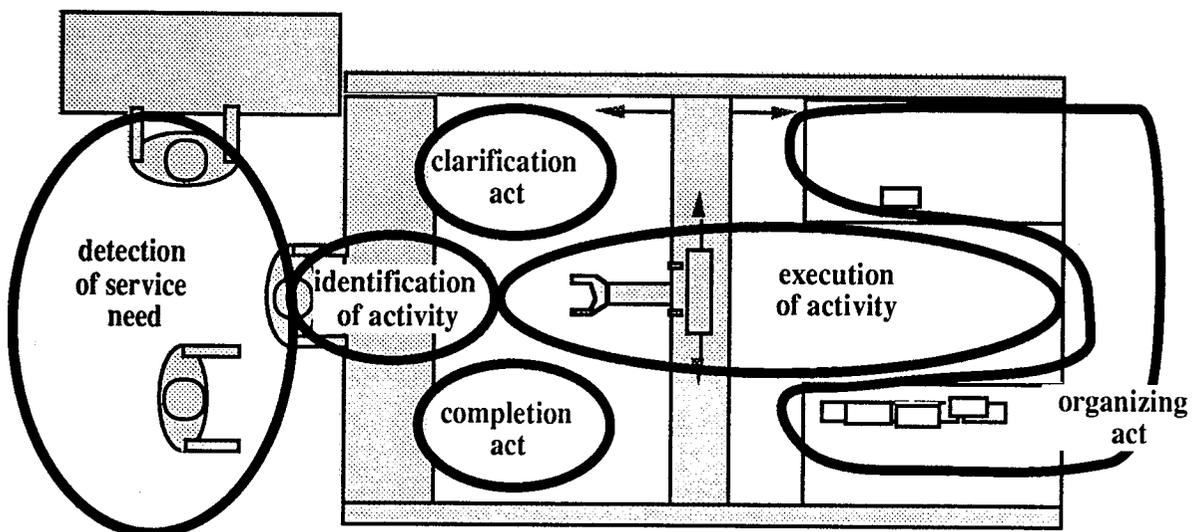
Figure 3: Example Operational Stages in the PC-Serbot Domain

- Completion Act

    Upon successful package manipulation, an agent can initiate acts to obtain acknowledgment from the customer that the transaction has been completed. For example, a completion act could be a simple closing customer dialog "Thank you, have a nice day." For customers picking up the package, the agent may effect a transfer of the package to the customer (e.g., place the package on the service counter and say "Here is your package") then issue a closing dialog (e.g., "Have a nice day").

- Organizing Activity

    During the course of an agent's activities, it can organize its work and state spaces in ways that allow it to perform faster or more efficiently. For example, considered placement of packages in a stocking area can significantly reduce the visual search and grappling that may later be required by an agent performing package retrieval.

These example agent action types illustrate the general type of functionality an agent may require when operating within the package counter domain. The next subsection describes two specific benchmark tasks on which an agent's performance can be evaluated.

3.6 Benchmark Task Problems

The package counter problem domain specifies the basic drop-off and pick-up transactions. A successful drop-off transaction is defined by the transfer of a package from a customer to the agent who indexes it for later reference and retrieval. A successful pick-up transaction is defined by the transfer of a package from the agent to a customer from a stocking area, or if the requested package does not exist, by the agent notifying the customer that the requested package does not exist. A transaction begins when a customer walks up to the service counter. A transaction is completed when the package in a transaction is received and stored, found and given to a customer, or determined not to exist.

Many specific benchmark tasks may be defined within the larger domain of package counter service robotics. This subsection details specifically the equipment and agent problem tasks used for the agent benchmarks. Section 4 will detail agent performance assessment techniques for this domain and related work in queueing theory and operations research.

3.6.1 Basic Package Counter Benchmark Task Description: BT1

The basic package counter benchmark task description, BT1, is intended to demonstrate and evaluate key elements of IAA while simplifying the required sensory effector implementation and support.

For the BT1 benchmark task, the apparatus shown in Figure 2 is used with the addition of a *customer service computer* which is placed on the service counter. This customer computer is used for all communication between customers and the agent.

In the initial configuration, there exist n packages in the stocking areas whose positions and identity are specified *a priori*. At any time, a customer may walk up to the service counter computer and enter their desire to perform one or more drop-off or pick-up transactions. A pick-up transaction requires that the customer enter a package identifier (e.g., their name or a receipt number). The customer may terminate a transaction at any time. Customer acceptance of a

retrieved package (provided via keyboard entry) defines completion of a pick-up transaction. Agent receipt, storage, and internal indexing of a package constitutes completion of a drop-off transaction. More than one customer may require service at any given time. The agent may use the service counter computer to acknowledge customer presence or requirements, and the agent may engage a customer in terminal dialogue as required to service their transaction. To simplify the testbed API and agent implementation, the customer must hand packages to the agent, and the agent must be able accurately and repeatably to pick and place packages without extended visuomotor support.

Aspects of this problem that challenge an IAA include:

- Asynchronous customer requirements
- Need for timely service
- Multiple customers introduce potential interleaving of operations
- Customer acceptance of package introduces objective credit assignment.

3.6.2 Extended Package Counter Benchmark Task Description: BT2

The extended package counter benchmark task, BT2, extends BT1 by adding sensorimotor capabilities for evaluating customer requirements and package handling.

In the initial configuration, there exist n packages in the stocking areas whose positions and identity *need not* be known *a priori*. In order to simplify perception for the testbed API builder, packages for task BT2 are monochromatic and easily distinguished by the color vision system provided by the testbed API. At any time, a customer may walk up to the service counter. The agent must use the testbed perception API to detect the presence of a customer at the counter, and determine whether a customer wants to perform one or more drop-off or pick-up transactions. A pick-up transaction requires the customer present the agent with a ticket of color identical to the package they are picking up. More than one customer may require service at any given time. The agent may engage in customer dialog to clarify transaction processing using the testbed API support for visual perception, sound generation, and robotic gestures. The agent can use the testbed API to visually identify package presence and location, and to perform the hand-eye coordination required to pick up a package. Unlike BT1, in BT2 the agent need not use a *customer service computer* for any communication between the customer and the agent. The agent in BT2 relies heavily upon the perceptual and motor skills afforded by the testbed API, and due the unreliabilities inherent in all such systems, benchmark task BT2 more heavily relies upon agent capabilities to overcome error, uncertainty and executional unreliability.

Aspects of task BT2 that challenge an IAA beyond that of BT1 include:

- Unreliabilities and uncertainties are introduced by use of perceptual and hand-eye systems
- Additional resource scheduling issues introduced by computational requirements of perception and hand-eye systems
- Richer human-agent dialog support that an agent can use to leverage its efforts
- More difficult implementational requirements.

3.7 Agent Evaluation in the Testbed

Figure 4 describes the operation of the testbed and benchmark scoring. As described in Section 3.6, the testbed agent API defines interfaces which provide physical state descriptions that the agent may use to model its environment. In turn, the agent controls the operation of these interfaces and the underlying equipment and computations. As the agent performs its tasks, the agent evaluation function computes the scoring metrics and it provides an overall agent score.

An agent is evaluated on the basis of its performance in executing package handling transactions for the benchmark tasks described in Section 3.6. As shown in Figure 4, testbed *agent evaluation* measures agent performance on benchmark transactions by computing scoring metrics (e.g., time per transaction, allocation of actions) and overall performance scoring; Section 4 discusses evaluation metrics, scoring and agent evaluation. Scoring metrics receive data from the execution of testbed agent API routines to compute resource allocation metrics such as "what percentage of time was spent in transit?".

The package handling transaction is a key unit upon which many agent performance metrics are based. To provide objective measurement of package transaction information, all customers are asked to enter their information at a testbed *customer terminal*. The *testbed customer API* defines customer interaction with this terminal. Package transaction information requires the user make entries at the customer terminal in order to initiate a transaction, abort a transaction, and complete a transaction. This transaction information is available to the agent for use in performing benchmark task BT1. In addition, the agent may use a text I/O stream provided by the testbed customer API to engage in additional customer dialog. For benchmark task BT2, agent use of customer terminal information should be severely penalized by the scoring metrics to encourage the use of sensorimotor actions provided by the testbed agent API.
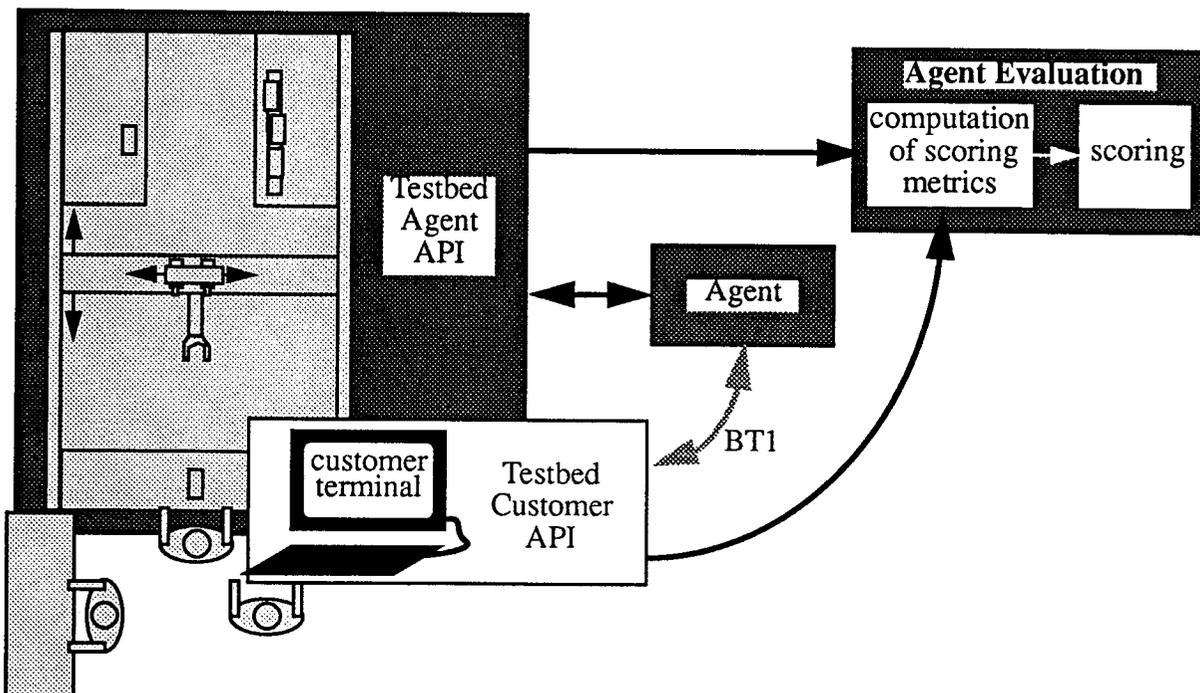
Figure 4: Operation of the Testbed and Benchmark Scoring

Figure 5 shows an example customer service terminal screen. There are two text buffers. In the text buffer labelled Transaction Registry, the customer or an observer enters transaction information required to perform evaluation and scoring. Human entered transaction initiation information includes:

- customer ID (e.g., their name)
- transaction type
    - drop-off
    - pick-up
        - package ID (e.g., the package color such as "purple").

Upon entering this information, the testbed customer API notes the time of transaction initiation and it assigns a unique transaction ID. To complete a transaction, as shown in Figure 5, the customer or observer enters when a transaction has been successfully completed or whether it should be aborted (e.g., because the agent took too long, or it got the wrong package too many times). Transaction information entered via the Transaction Registry screen is made available to the agent via the testbed customer API, though as noted earlier for benchmark task BT2, agent use of this terminal information should be severely penalized by the scoring metrics.

The agent does not affect the dialog in the Transaction Registry. As shown in Figure 5, a Customer Service Terminal buffer provides the agent with bidirectional text I/O for which they may write messages to the customer and read customer responses. The agent is responsible for the structuring, processing, parsing and control of this textual customer interface. In the examples shown in Figure 5, the agent is verifying the type of desired customer transaction,

```
┌─────────────────────────────────────────┐
│  Customer Service Terminal                │
│                                           │
│  Philip, to verify, you have a package    │
│  to pick up?                              │
│  Enter Y or N: Y                          │
│                                           │
│  Philip, Do you want this package?        │
│  Enter Y or N: _                          │
│                                           │
│                                           │
├─────────────────────────────────────────┤
│  Transaction Registry                     │
│                                           │
│  ***** MAIN TRANSACTION MENU *****        │
│  (1) Initiate transaction                 │
│  (2) Abort transaction                    │
│  (3) Complete transaction                 │
│  (4) List active transactions             │
│  Enter choice (1-4): _                     │
└─────────────────────────────────────────┘
```
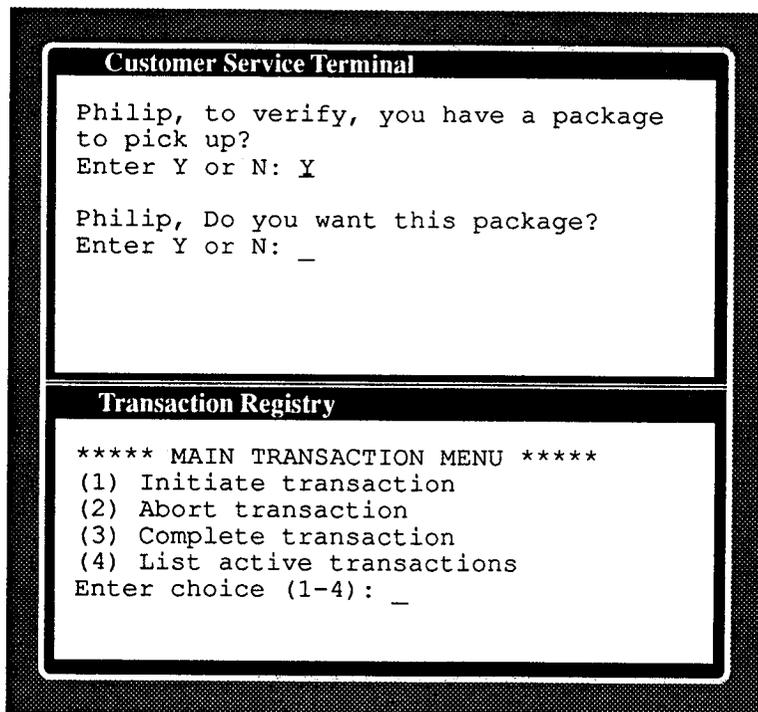
Figure 5: Example Package Counter Terminal Screen

and the agent is asking for verification that a package it is holding it the desired customer package. As noted earlier, this terminal usage should be severely penalized by the scoring metrics.

Evaluating a reactive planner in the PC-Serbot testbed involves

- Implementing a package handling agent using the PC-Serbot testbed API described in Section 5 to execute package pick-up and drop-off tasks (transactions)

- Executing the package handling agent within the testbed on sequences of pick-up and drop-off transactions

- Computing and evaluating agent transaction scoring metrics and benchmarks described in Section 7.

The resulting agent evaluation metrics provide key data that can be used to examine how an agent performed within the testbed and the reasons underlying its performance.

3.8 Agent Task Support

By appropriately controlling the sensorimotor systems provided by the PC-Serbot API, an intelligent agent can effectively execute the two main tasks (transactions) within the package counter testbed benchmark domain. The testbed API must provide the agent executing benchmark tasks with adequate means for interfacing with customers and packages. The testbed agent API, detailed in Section 5.5, provides for agent control and data acquisition using

- positions and operating traits for the robotic degrees of freedom (e.g., arm, end effector, gantry, head)

- perceptual systems providing information to assess package and customer state.

Figure 6 shows how the testbed API (in the grey outer ring) provides agent interfaces to the customer and package environment.

As described in Section 3.6, agent benchmark task BT1 has reduced API support requirements. Of the package interfaces shown in Figure 6, BT1 only requires agent interface calls to control the arm end effector. These calls must allow the arm to move to a known point in space, and the end effector may be opened or closed to grab or release an object. In addition, these calls must be able to indicate whether an object is in the end effector (i.e., it must answer "is something in my hand?"). Of the customer interfaces shown in Figure 6, BT1 only requires a textual agent interface to the customer. This "package counter terminal" interface provides the agent with a text I/O stream for which they may write messages to the customer and read customer responses. The agent is responsible for the structuring and control of this textual customer interface.

Agent benchmark BT2 requires substantially more complex sensorimotor support from the testbed API for customer and package interactions. As shown in Figure 6, BT2 requires that an agent be provided with additional customer interfaces. Calls to play sound files are provided to allow the agent to control customer dialog and desired actions. For example, the agent may use a prestored sound file to ask the customer to "Please put the package in my hand." Text to speech generation may alternatively be used with no significant change to the interface. In addition, testbed API calls to determine customer position and state are required if the agent is to visually interact with customers. For the purposes of BT2, this visual interface must be capable of visually assessing customer presence and position at the service counter. To support dialog with a
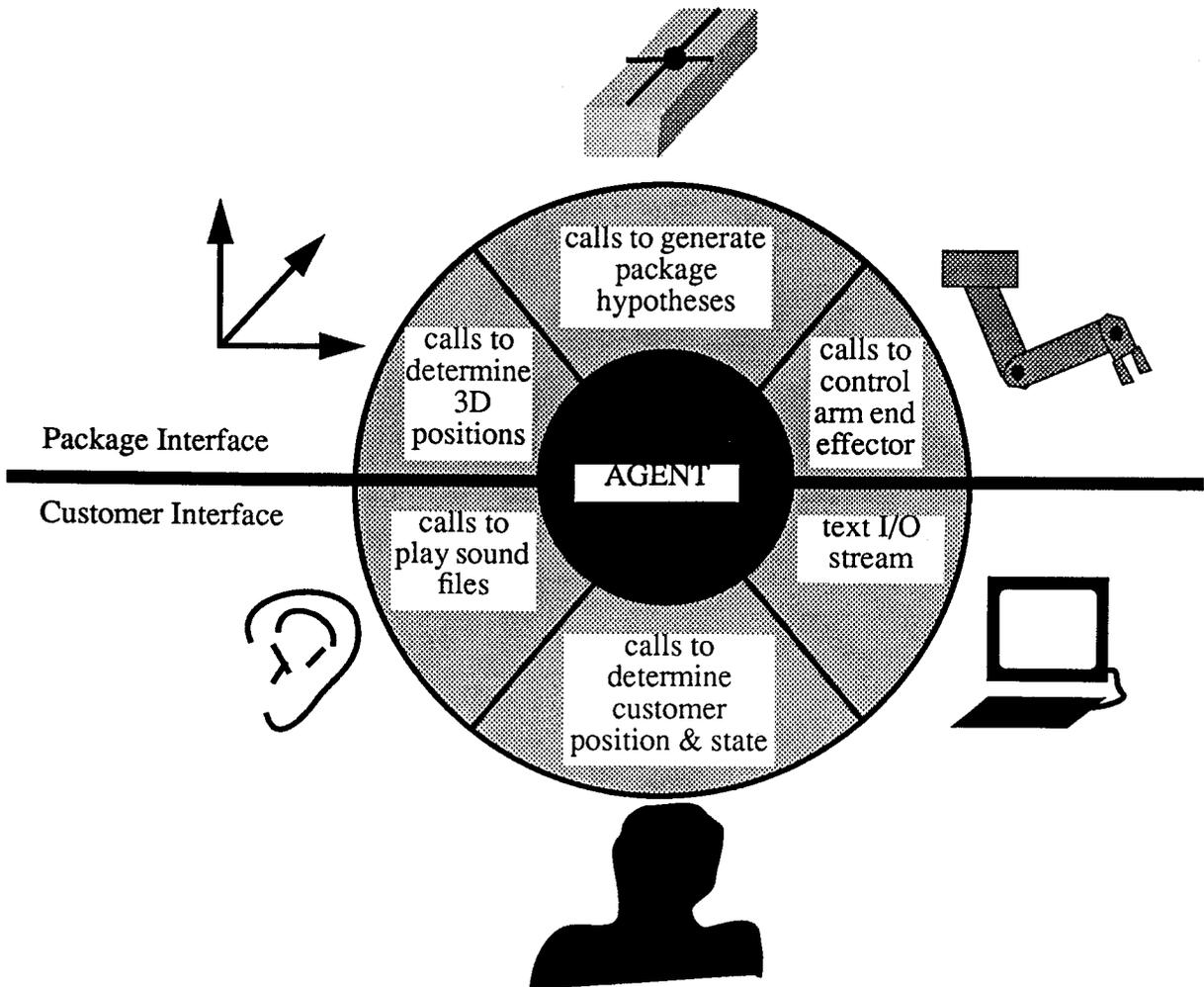
Figure 6: PC-Serbot Agent and Interfaces to its Environment.
The testbed API in the grey circle provides interfaces between the agent and
the packages and customer.

customer, the visual interface must also provide the agent with the ability to determine the answer to an interrogative. For example, an agent may utter "Do you have a package? Please nod yes or no," and the visual customer interface must be capable of detecting whether a yes, no or neither has been detected. The agent still has the textual customer interface if they deem it necessary, though for BT2, the evaluation metrics should penalize agent use of the textual interface.

The testbed API provides CommonLISP function calls that define the logical interface between the intelligent agent being evaluated and the testbed package handling environment. Section 5.5 provides a detailed description of these testbed API calls and their arguments.

3.9 Why the PC-Serbot Domain?

Among the contributions of this project has been the development of a canonical problem that both tests and demonstrates key elements of intelligent agent architectures (IAA). The use of such canonical problems in the planning community, for example the "blocks world," has done much for describing and extending their abilities. Since IAA provide unique problem solving capabilities, the canonical PC-Serbot problem and evaluation metrics described in this report can form the basis of future IAA research and development.

During this project, many IAA problem domains were considered (e.g., an agent digging for metal cubes in gravel, an agent stacking loads on toy trains). As they were considered, the desired criteria listed in Section 3.2 were developed. Of the considered domains, the PC-Serbot domain inevitably proved to best fulfill the desired criteria. In particular, PC-Serbot was found to best satisfy the three primary conflicting aims:

- exercising key traits of IAA in a problem domain with strong ties to more traditional canonical planning problems
- usefulness, realism and generality of the problem domain
- implementation feasibility.

The first characteristic tends to drive the benchmark domain to more academic, controlled domains in which IAA issues may be isolated while simplifying implementation. The second characteristic tends to drive the domain to overly application-specific problems of great implementation difficulty. The last characteristic tends to drive the domain to relatively trivial solutions that do not well achieve the other two criteria. The PC-Serbot problem domain represents our best attempt to derive a canonical problem domain that best balances these conflicting forces while still achieving significant progress in all three characteristic dimensions.

4 TESTBED AGENT PERFORMANCE EVALUATION

Benchmarking the performance of an agent requires that computational performance metrics be defined and measured. The previous section described a specific agent problem-task domain; this section describes how to assess the performance of agents operating within this domain.

We distinguish between testbed evaluation metrics and scoring metrics. *Evaluation metrics* provide narrow and specific performance assessments that describe aspects of agent effectiveness. An example evaluation metric might be the average or worst case package transaction execution time. A *scoring metric* provides a weighting of these evaluation criteria in order to provide a subjectively satisfying summary judgment of agent performance. Scoring metrics are intended to provide simple comparisons between competing agent construction methodologies. These scoring metrics may then be used to judge the relevance of individual evaluation metrics, the effectiveness of an agent, and the appropriateness of the relative weighting used by the scoring metric.

4.1 Testbed Evaluation Metrics

This subsection describes objective and computable metrics that can be used to judge aspects of agent performance within the package handling scenario described in Section 3. These evaluation metrics address how and effective agent performs on component actions within the package transaction processing problem domain.

4.1.1 Objective Evaluation Criteria

4.1.1.1 Temporal Metrics

The following evaluation metrics describe the time spent executing component transaction steps. As standard in queueing theory, these metrics may consider the best, average and worst case times. The best time is an indicator of how well an agent can expect to perform that function. The average time is a good indicator of expected overall performance. The worst case time is useful for identifying degenerative cases in which agent performance may be unacceptable.

- wait for service time (T_{ws})

  This metric describes how long a customer must wait before the agent begins their transaction. This is a standard evaluation metric used for transaction analysis in queueing theory.

- package transaction completion time (T_{tc})

  This metric describes the time to taken by an agent from the start of a transaction to its completion. This is also a standard evaluation metric used for transaction analysis in queueing theory.

- package stack manipulation time (T_{sm})

  This metric describes the time spent by an agent in physically picking up, gripping, releasing or moving packages in its workspace. A related metric is the distance over which packages are transported, though time is perhaps a more descriptive and useful metric of overall performance.

- **perception processing time** (T_{pp})

    This metric describes the time spent by an agent in computing perceptual properties of the package domain. These perceptual processing times can be divided into more detailed categories: for example, time spent visually identifying customer or package identity or position. Another useful metric that can be computed is the delay from when visual computations are requested and when they are received. As the complexity of visual tasks increases (e.g., find edges to find objects), this processing time may increase, and an effective agent can explicitly employ strategies to reduce this latency.

- **human interchange time** (T_{hi})

    This metric describes the time spent by an agent in dialog with a customer in order to define transaction requirements or properties. For example, this metric can include the time required for an agent to identify whether a pick-up or drop-off transaction is desired. Further, the metric can be divided into more detailed categories that note differences between human interchange time for pick-up and drop-off transactions. For example, a pick-up transaction engenders the need for an agent to enter into a dialog to identify the desired package.

4.1.1.2 Modelling Metrics

Modelling metrics measure objective characteristics of the package environment that can affect agent performance, and for which temporal metrics may not provide stable indicators of performance or agent response to problem complexity.

- **object model completeness** (T_{mc})

    In many cases, the performance of an agent is directly related to the degree to which they have an explicit description of the problem domain. For example, an agent that knows all of the positions of packages in its working area may be able to perform more efficiently than an agent that operates without state. Without considering agent-specific modelling, it is possible to provide objective metrics that describe the knowledge state of an agent. For example, the bounded ratio $\frac{IDed\,boxes}{NUM\,boxes}$ is a useful description of agent state that affects its performance.

- **perceptual accuracy** (T_{pa})

    Perception is a key factor in agent effectiveness. Perception is subject to error, and the manner in which an agent handles these errors has significant effects on agent competence and the time spent performing tasks.

    In the package domain, it is useful to have an evaluation metric describing package identification accuracy. One method for quantifying this metric describes the accuracy of a perception system as the ratio of incorrect perceptual assessments to the total number of perceptual assessments that have been performed by an agent. This may be measured at the end of a pick-up transaction by determining whether the desired object was correctly identified by the perception subsystem. For drop-off transactions, this may be measured at time of package-object model acquisition by checking to see whether any other acquired objects match the newly acquired object.

    It may also be useful to have customer identification accuracy evaluation metrics. For example, when multiple customers may be simultaneously served, customer perception accuracy may be measured at the time when agents address a given customer: if the wrong

customer is identified, that may then be registered. When more specific operational properties are known about the perceptual system and its operation, more accurate measurement methods may be defined.

4.1.2 Controlled Factors

Within the package transaction agent domain described in Section 3, several factors may be quantified and controlled. These factors affect the performance of agents, and they can be used to distinguish important operational differences between agent methodologies.

- **number of packages**

  As the number packages in the agent workspace increases, the possible complexity of transaction processing may also increase. For example, an agent that maintains no state requires looking anew for each package; it thus will have $O(n)$ performance where n is the number of packages in the work space. Conversely, an agent which maintains package position information can achieve $O(1)$ performance when packages do not move. When packages are moved in the work space without the knowledge of the agent (*e.g.*, by a package falling or a person moving it), an effective reactive agent will respond, and its performance may be sensitive to the number of packages.

- **probability distribution for customer arrival**

  The temporal evaluation metrics described earlier are affected by the probability distribution for customer arrival. Classical queueing theory has much to say about how these distributions may be modelled, and we do not consider them here. However, using such a model, it is possible prior to an agent evaluation experiment to generate customer arrival times *a priori* and this may be used to schedule when customers should approach the counter.

- **initial package model completeness**

  The performance of an agent may also be significantly affected by the completeness of its start-state representation of packages in its workspace. For example, when unknown packages are in the workspace, the agent may need to visually scan the storage areas, identify packages and their locations, and perhaps pick up and reorganize packages in the workspace.

  The completeness and accuracy of the vision system modelling of packages may also be used to describe an element of complexity that an agent must address. For example, a given set of packages may be used in testing an agent, and the performance of the vision system in learning and recognizing these packages may be measured. One method could learn n views of each object (*e.g.*, all of its sides at long, medium and near range), and then, a statistical model of recognition accuracy may be constructed (*e.g.*, a confusion matrix between object views, or expected accuracy for a randomly selected object and view). The completeness of the vision system model determines how resourceful an agent must be in overcoming inaccuracies in visual system performance.

4.1.3 Uncontrolled Factors

Physical scenarios have sources of agent complexity that may not be suitably modelled or controlled. An effective agent may be able to overcome these problem sources without an explicit model of them by applying general strategies (*e.g.*, reactive control or planning). Example uncontrolled factors for the package testbed scenario described in Section 3:

- grappling errors (visuomotor sources)
- effects due to position and appearance of customers
- unmodelled vision system errors (due to lighting, algorithms, etc.)
- kinesthetic errors.

4.2 Testbed Scoring Metrics

A *scoring metric* weights evaluation criteria in order to provide simple comparisons between competing agent methodologies. These scoring metrics may then be used to judge the relevance of individual evaluation metrics, the effectiveness of an agent, and the appropriateness of the relative weighting used by the scoring metric.

An agent score numerically assesses how well an agent performs testbed tasks in a "desired" manner. At the simplest, an evaluation criteria may directly score an agent (*i.e.*, average transaction time). Alternatively, multiple evaluation criteria may be combined when agents need to balance competing performance goals (*e.g.*, speed versus accuracy).

Many methods may be used to combine multiple evaluation metrics into an overall score. A straightforward approach is to sum weighted evaluation metrics into a single score. When evaluation metrics produce bounded and normalized values (e.g., values between ± 1), the weighting coefficients simply specify the relative contribution or importance of each metric. Ratios between evaluation metrics also provide a useful way to create scoring mechanisms which capture relative agent performance. More complex score summation techniques may be used — for example, evaluation metric coefficients that nonlinearly incorporate other metric values — though interpretation of these more complicated scoring methods may be less readily apparent. The proposed method of scoring, then, is to compute a weighted sum of these general component scoring criteria.

Three general scoring criteria may be useful in understanding agent performance in the PC-Serbot application domain:

1. Package handling performance
2. Efficiency
3. Responsiveness.

Each of these component scoring criteria are now described in turn.

4.2.1 Package Handling Performance

Basic agent package handling performance in the PC-Serbot domain is a key determinant of that agent's overall competence. An agent that cannot reliably perform basic package handling transactions will be incompetent in executing the package handling benchmark tasks described in Section 3.4. Beyond this basic capability, a key differentiator between agent methodologies will

be the reliability and efficiency of their package handling performance. For example, an agent that can adapt and overcome unexpected variations in the package domain — a package falls, or is moved by a person, or perception has detection glitches — may be expected to operate more reliably; but perhaps at the cost of more computation or greater time lags.

Two main components that can be used to score package handling performance are

1. Package perception

Asks: how well is the visual perception system detecting, identifying and locating packages in the environment? Related evaluation metrics are: perception processing time, object model completeness, perceptual accuracy, number of packages (both total to date and total in current workspace), and initial package model completeness. Using some of these metrics, a package perception scoring scheme is:

$$S_{PHP} = \frac{T_{pp}}{N} + T_{pa} - \frac{n}{N}T_{mc}$$

where S_{PHP} is the score for package handling perception (low score bad, high score good), n is the number of packages currently in the workspace, N is the total number of package transactions processed thus far, and the other metrics are described in Section 4.1.

2. Package grappling and transport

Asks: how well is the package pick-up, manipulation and transfer doing? Related metrics are package stack manipulation time, perceptual accuracy and object model completeness. A package perception scoring scheme is:

$$S_{PHG} = \frac{T_{sm}}{N}$$

4.2.2 Efficiency

Transactional analysis in queueing theory often examines the average throughput rate as a measure of efficiency. This scoring provides a good way to differentiate between agents that employ a more globally optimal problem solution strategy from agents that may not consider the effect of local actions on overall performance and efficacy. Given the same resources and workloads, an agent with higher average throughput may be considered more efficient than another agent with less throughput. A way to score agent efficiency using average package transaction completion time (throughput) is:

$$S_E = \overline{T_{tc}}$$

Agent transaction processing throughput depends on the processing demands and on how an agent responds to them. Attention to the effect of controlled factors is important in understanding throughput-based scores (see Section 4.1.2). Also, throughput analysis is anecdotal until a statistically significant number of runs is made. Care must also be shown since degenerate cases can also have a large effect; one agent may have very good average throughput, but on one degenerate case it may never complete a transaction and its throughput will halt.

4.2.3 Customer Responsiveness

A common scoring method used in queueing theory looks at the responsiveness of a server to transaction demands. When transaction load demands vary, this scoring provides a good way to differentiate between agents that can balance throughput and responsiveness, and agents that may not as effectively consider the global effect of local actions. A way to score agent responsiveness to customer transaction demands using average customer wait for service time is:

$$S_R = \overline{T_{ws}}$$

4.3 Computing the Metrics

Support must be provided to compute the evaluation metrics. For the timing metrics, testbed API calls can be supplemented to determine when and which metrics are updated. Measuring modelling metrics requires that model representations be extended to include incremental variables used to update the metrics. Section 4.1 described methods for computing metrics.

As in any real-time system, the metric and scoring computations consume resources which can alter agent operation. The metrics described in Section 4.1 are specifically intended to provide useful and relatively straightforward computation that is not likely to affect agent execution. Even so, it is good practice to place metric computation code within the Testbed API which does not impose the hard real-time requirements that other hardware subsystems may impose (*e.g.*, the gantry motor controller or vision system processor).

5 THE PC-Serbot TESTBED API

This section describes the PC-Serbot Testbed Application Programmer's Interface (testbed API) that an agent may use to effectively perform the benchmark tasks BT1 and BT2 (described in Section 3.6). This testbed API provides a specific software interface and semantics that define what an agent may do and sense within the testbed domain. Section 6 describes implementation of the testbed API.

Section 3.8 discussed agent task support required by benchmark tasks BT1 and BT2. The next section provides an overview of the specific sensor and effector resources an agent may apply to these benchmark tasks. Section 5.3 reviews issues that affect the level at which the API provides an abstraction of the task domain. Section 5.5 then provides a detailed description of the testbed API which provides a CommonLISP/CLOS software interface to the agent and agent builder.

5.1 Testbed Agent Resources for Benchmark Task 1 (BT1)

This subsection overviews the testbed resources available to an agent performing benchmark task BT1 that was described in Section 3.6.1. Detailed descriptions of the CommonLISP/CLOS software interfaces are provided in Sections 5.4 and 5.5.

5.1.1 The Gantry

As shown in Figure 2, the gantry is a two-dimensional X-Y translation device for moving the physical agent within its work space which includes the package stocking areas and the customer service counter. The agent control software is provided with the ability to initialize, move, query position, and rehome the gantry device.

5.1.2 The Arm and Gripper

A robot arm and gripper is mounted beneath the gantry to allow the agent to manipulate packages. The agent is provided with the ability to initialize, move (in joint and cartesian endpoint space), open and close the gripper, query joint and gripper positions, detect when an object has been placed in the gripper, and rehome the arm and gripper devices.

5.1.3 Testbed Customer API

Section 3.7 provided a description of the customer terminal which allows users to enter textual information. The *testbed customer API* defines customer interaction with this terminal.

To ensure accurate agent scoring, the `Transaction Registry` text buffer allows a customer or an observer to enter transaction information required to perform evaluation and scoring. Package transaction information requires the user make entries at the customer terminal in order to initiate a transaction, abort a transaction, and complete a transaction. The transaction registry API specifies the customer interface, the data structures and the agent calls to access these data structures. As noted earlier, in order to allow evaluation metrics to penalize an agent for use of this textual customer input (e.g., for BT2), agent access to transaction registry information is available to the agent scoring mechanism.

The agent does not affect the dialog in the `Transaction Registry`. As shown in Figure 5, a `Customer Service Terminal` buffer provides the agent with bidirectional text I/O for which they may write messages to the customer and read customer responses. The agent is responsible for the structuring, processing, parsing and control of this textual customer interface. As with the transaction registry, agent access to customer service terminal information is available to the agent scoring mechanism.

5.2 Testbed Agent Resources for Benchmark Task2 (BT2)

As described in Section 3, agents performing benchmark task BT2 require extended support and interfaces to detect and interact with packages and customers. This subsection overviews additional testbed API resources available to an agent. Detailed descriptions of the CommonLISP/CLOS software interfaces are provided in Section 5.5.

As described in Section 3.6.2, for BT2, the agent is responsible for detecting and servicing customer package transactions. To do this, an agent must be able to perceive customer presence at the service counter and customer transaction requirements, and it must effect customer package transfers (i.e., take or give a package). The testbed API provides several resources that an agent may apply to perform these functions. An agent is provided with primitive sensory actions which include a stereo system to determine the relative position of environmental objects (e.g., such as a person standing at a counter), a color vision system to characterize environmental objects (e.g., the guy with the red shirt), and an active camera head to control gaze (e.g., to scan the service counter for customers). In addition, two integrated actions (behaviors) are available for agents that wish to interact with the testbed: a customer detection behavior and a visual tracker. The customer detection behavior controls head gaze and vision sensors to scan the service counter to find customer position. The visual tracker maintains the cameras' gaze upon an environmental object; example uses include monitoring customer position even while undergoing relative movement, and detecting customer "yes or no" head nods in response to agent questions (via detection of cyclic head tracking). The testbed API also includes a sound player which an agent may use to engage in dialog with the customer by playing pre-recorded sound files (e.g., "Please hand me the package," "Is this your package?").

For the benchmark task description in Section 3.6.2, colored packages may be characterized in several ways. Color vision measurements may be used to identify pick-up slips and to generate package detection and position hypotheses. In order to pick up a package, hand-eye coordination can use stereo to better characterize package size, shape and location. When an active camera head is available to change camera gaze, visual search for packages is possible, and visual tracking can be used to maintain visual attention while the package, gantry or effector is in transit.

The next subsections briefly describe the additional sensorimotor capabilities available to an agent performing benchmark task BT2.

5.2.1 Stereo and Motion Measurements

In order to detect and describe packages and customers in the testbed environment, the testbed API includes a real-time stereo and visual motion measurement capability.

The testbed API includes a monochromatic stereo vision system that can determine the relative distance of environmental objects relative to the agent. This stereo system produces depth

measurements which are either relatively confident and accurate, within a small percentage of distance to object, or for which inadequate information is available to produce a reliable depth estimate. The stereo system produces these depth measurements in real-time. The agent is responsible for invoking and controlling the operation of the stereo system per the API defined in Section 5.5.2.

In addition to stereo depth measurements, a visual motion measurement API is provided to the agent for perceiving testbed state. As with the stereo system, sparse visual flow is provided where measurements provide confident results, and visual flow is assumed to be computed in real-time. Such visual flow is particularly useful for discriminating a moving object against a stationary background (such as a customer walking up to the service counter). The agent is responsible for invoking and controlling the operation of the visual motion system per the API defined in Section 5.5.2.

5.2.2 Color Vision Measurements

There are two primary functions of the testbed color vision API: measure properties of a known object part (e.g., "what color is the object I am looking at?"), and find a desired object characterized by its color (e.g., "do I see a red package?"). Like the stereomotion vision systems, the color vision system is assumed to operate in real-time. Calls to the color vision system to measure (learn) a known object returns a color object description that may later be used by the color vision system to find the object again at a later time. The agent is responsible for invoking and controlling the operation of the color vision system per the API defined in Section 5.5.3.

5.2.3 The Active Camera Head

To allow the agent to control visual attention, an active camera head is mounted atop the gantry robot. The active head is assumed to control color and stereo camera pan and tilt (i.e., elevation and azimuth). For example, using the active head and the color vision system, an agent may visually scan the testbed area to find a package. The agent is responsible for invoking and controlling the operation of the color vision system per the API defined in Section 5.5.2.

5.2.4 The Sound Player

To allow the agent to communicate easily with a customer and to engage in more natural dialog, a sound player is provided so that an agent can "utter" pre-recorded sound files (e.g., "Please hand me the package." An alternate sound support mechanism consistent with this interface would be the use of a text-to-speech synthesizer that allows an agent to generate verbal utterances from text strings. In either case, the agent is responsible for invoking and controlling the operation of the sound player per the API defined in Section 5.5.4

5.2.5 The Tracker

Using the primitive actions described above, an agent can directly synthesize stereomotion and active head control to perform visual tracking. Alternatively, an integrated action (behavior) is provided for visual tracking. Such visual tracking is often implemented as an integrated action since the real-time constraints its faces generally require a highly integrated implementation to

provide adequate performance. The agent is responsible for invoking and controlling the operation of the tracker per the API defined in Section 5.5.5.

5.2.6 Customer Detection Behavior

Using the primitive actions described above, an agent can directly synthesize action sequencing to perform detection of customers at the counter. Alternatively, an integrated action (behavior) provides a higher level functionality that an agent may use to perform customer detection. The customer detection behavior moves the gantry robot to a good vantage point, and it then visually scans the service counter looking for a customer. If a customer is found, it then scans up to their head. At that point, the agent may perform other actions such as invoke the tracker (to allow for simple head gesture detection), measure color properties of the customer head or torso for later reacquisition, or initiate a dialog to determine the desired customer transaction. The agent is responsible for invoking and controlling the operation of the customer detection behavior per the API defined in Section 5.5.5.

5.3 Atomic Instantaneous Actions vs. Integrated Actions (Programmed Behaviors)

When defining an agent testbed API, it is important to consider the appropriate granularity and abstraction for agent actions. The classical reactive agent approach is to design API actions that are purely atomic and mutually exclusive (sometimes called primitive actions). These primitive actions may be considered independent and the agent and agent designer perform all primitive action sequencing to perform a task. Alternatively, more integrated actions may be constructed which control primitive actions. These integrated actions locally integrate the sequencing of primitive actions to perform more abstract functions. For example, "pick up object X" is an integrated action that would sequence and control multiple primitive actions (e.g., move the arm, pre-form the wrist, close hand, monitor forces). These integrated actions have also been called *behaviors* or *programmed behaviors*.

The testbed agent API is largely comprised of atomic actions, though several programmed behaviors have also been included to provide agents with some higher level capability. Below are some general characteristics of both action types.

Properties of atomic mutually exclusive actions:
- Advantages
  - *provides an agent with complete control of sequencing: can obtain high degree of serendipity, opportunism, and emergent assemblages of exhibited behavior*
  - *simpler, more generic action specifications*
  - *resource allocation completely within control of the agent*
- Disadvantages
  - *need for atomic mutual exclusivity generally allows only very low level interfaces to resources (e.g., sensory, effector and the computing systems)*
  - *agent coordination for low-level interfaces can have high complexity*

- *because of possible interference and contention, agents requiring atomic exclusivity may not be able to use integrated actions that coordinate multiple atomic actions (e.g., integrated hand-eye actions to pick up an object)*

Properties of integrated actions (programmed behaviors):
- Advantages
  - *integrated actions allow agents more flexibility in choosing the granularity and complexity at which they control actions and perception*
  - *integrated actions have been defined that may be of use to an agent (e.g., visual tracking)*
  - *can implement an integrated action for which a phenomenological model may not be available or feasible (e.g., a recipe for action vs. a reasoned course of action)*
  - *modular development and testing of integrated actions simplifies development of agent capabilities*
  - *can characterize integrated actions by their performance and other traits (e.g., overall time required, risk of failure, elapsed time in action)*
- Disadvantages
  - *when an integrated action does not provide for control of all factors relevant to an agent, an agent can be severely limited in its ability to control actions to exhibit serendipity, opportunism, emergent capabilities and perform subtask deconfliction.*

5.4 PC-Serbot Testbed Customer API

This section describes the agent API for the testbed customer interfaces to the Customer Service Terminal and Transaction Registry that were described in Section 3.7.

5.4.1 Transaction Registry API

As was described in Section 3.7, the Transaction Registry allows a customer or testbed observer to accurately define transaction initiation, identification and termination. This transaction information is used by the testbed scorer to evaluate agent performance.

5.4.1.1 Methods and Functions

```
(in-package :tcx)
(export '(
    CLEAR-TRANSACTIONS
    INITIATE-TRANSACTION
    ABORT-TRANSACTION
    COMPLETE-TRANSACTION
    LIST-ACTIVE-TRANSACTIONS
    LIST-ALL-TRANSACTIONS))

(defun CLEAR-TRANSACTIONS ()
  "Clears all transactions in the registry."
```

```
(defun INITIATE-TRANSACTION (&key  (start-time (get-universal-time))
                                   (default-custID ""))
    "Registers a new transaction."


(defun ABORT-TRANSACTION (&key  (default-custID "")
                                (end-time (get-universal-time)) )
    "Registers abnormal termination of an in-progress transaction."


(defun COMPLETE-TRANSACTION (&key   (default-custID "")
                                    (end-time (get-universal-time)) )
    "Registers successful completion of an in-progress transaction."


(defun LIST-ACTIVE-TRANSACTIONS ()
    "Returns all in-progress transactions."


(defun LIST-ALL-TRANSACTIONS ()
    "Returns all transactions registered to date."
```

5.4.1.2 CLOS Objects and Structures

```
(defclass TRANSACTION ()
    ((custID :initarg :custID :accessor custID :initform nil
            :documentation "The unique customer ID initiating the transaction.")
     (transactionID :initarg :transactionID :accessor transactionID :initform nil
            :documentation "The unique transaction ID.")
     (transaction-type :initarg :transaction-type :accessor transaction-type :initform nil
            :documentation "The transaction type (1 for pick-up, 2 for drop-off")
     (packageID :initarg :packageID :accessor packageID :initform nil
            :documentation "The unique package ID.")
     (start-time :initarg :start-time :accessor start-time :initform nil
            :documentation "Universal time when transaction was initiated.")
     (end-time :initarg :end-time :accessor end-time :initform nil
            :documentation "Universal time when the transaction was completed or terminated.")
     (completion-status :initarg :completion-status :accessor completion-status :initform nil
            :documentation "Completion status (:active, :abnormal, :successful).")
     (performance-metrics :initarg :performance-metrics
            :accessor performance-metrics :initform nil
            :documentation "Performance metrics in object maintained by the scorer.")
     ))
```

5.4.2 Customer Service Terminal API

As was described in Section 3.7, the Customer Service Terminal provides a bi-directional textual I/O stream that an agent may use to engage in customer dialog.

5.4.2.1 Methods and Functions

```
(in-package :tcx)
(export '(
    WRITE-TO-CUSTOMER
    READ-LINE-FROM-CUSTOMER
    READ-FROM-CUSTOMER
    PROMPT-CUSTOMER
    ))
```

```
(defmethod WRITE-TO-CUSTOMER ((text string))
  "Writes a string to the *CUSTOMER-SERVICE-TERMINAL* emacs buffer."
```

```
(defmethod READ-LINE-FROM-CUSTOMER (&key (flush nil))
  "Reads a string from the *CUSTOMER-SERVICE-TERMINAL* emacs buffer.
   FLUSH of t will flush all prior entered data prior to the read."
```

```
(defmethod READ-FROM-CUSTOMER (&key (flush nil))
  "Reads an s-expr from the *CUSTOMER-SERVICE-TERMINAL* emacs buffer.
   FLUSH of t will flush all prior entered data prior to the read."
```

```
(defmethod PROMPT-CUSTOMER ((prompt string)
    &optional (result-type 'string))
  "Writes PROMPT and reads user input of type RESULT-TYPE from the
   *CUSTOMER-SERVICE-TERMINAL* emacs buffer. Repeats prompting until data is entered
   of type RESULT-TYPE."
```

5.4.2.2 CLOS Objects and Structures

The Customer Service Terminal API has no object or structure definitions.

5.5 PC-Serbot Testbed Agent API

This section describes the CommonLISP calls and CLOS Objects and Structures developed under this project that support the PC-Serbot scenario and capability described in earlier sections.

5.5.1 Gantry Robot API

5.5.1.1 Methods and Functions

```
(in-package :tcx)
(export '(
        GANTRY-INIT            ; function to get GANTRY CLOS object
        GANTRY                 ; initializes the PRISM system
        RESET                  ; resets the state of the GANTRY system
        SHUTDOWN               ; shuts down the GANTRY SERVER and releases it

        GET-JOINT-POSITIONS    ; query returns GANTRY joint status and configuration
        SET-JOINT-POSITIONS    ; command changes GANTRY position and parameters
        RELEASE                ; command to open the GANTRY gripper
        GRAB                   ; command to close the GANTRY gripper
        GET-CARTESIAN-POSITION ; query returns the GANTRY end effector position
        SET-CARTESIAN-POSITION ; command to change the GANTRY end effector position

        ))
```

```
(defun GANTRY-INIT (&key (timeout-in-msecs 30000)

                         (gantry-name "gantrycontrol")

                         (reset t))
```
"This function initializes TCX communication with the gantry and returns
a CLOS/LISP gantry communications object."

```
(defun GANTRY ()
```
"This function returns the current gantry CLOS object."

```
(defmethod SHUTDOWN ((state gantry)

                     &key (await-completion t)

                          (timeout-in-msecs 10000) )
```
"This method rehomes the gantry and shuts it down in an orderly way."

```
(defmethod GET-JOINT-POSITIONS ((state gantry)

                                &key (timeout-in-msec 10000) )
```
"Query returns the relative axis position of the gantry's degrees of freedom in the
GanrtyJointPosition class (X and Y are specified in millimeters and joint angles are specified
in radians)."

```
(defmethod SET-JOINT-POSITIONS
                ((state gantry)
                &key (await-completion        t)
                        (timeout-in-msec      10000)
                        (Xpos                 *GANTRY-DONT-CARE*)
                        (Ypos                 *GANTRY-DONT-CARE*)
                        (elbow-angle          *GANTRY-DONT-CARE*)
                        (WristRoll-angle      *GANTRY-DONT-CARE*)
                        (WristPitch-angle     *GANTRY-DONT-CARE*)
                        (ToolRoll-angle       *GANTRY-DONT-CARE*)
                        (gripper-X            *GANTRY-DONT-CARE*) )
"Command moves the gantry to achieve the desired relative gantry positions."
```

```
(defmethod RELEASE ((state gantry)
                &key (await-completion t)
                        (timeout-in-msec 10000))
"Opens gripper. Currently does not return any acknowledgement."
```

```
(defmethod GRAB ((state gantry)
                &key (await-completion t)
                        (timeout-in-msec 10000) )
"Closes gripper until the motor stalls. Currently does not return any
acknowledgement."
```

```
(defmethod GET-CARTESIAN-POSITION ((state gantry)
                                        &key (await-completion t)
                                        (timeout-in-msec 10000) )
"Query returns the wrist endpoint position and orientation in Cartesian space
in the GantryCartPosition CLOS object."
```

```
(defmethod SET-CARTESIAN-POSITION
                ((state gantry)
                &key (await-completion t)
                        (timeout-in-msec 10000)
                        (xPos *GANTRY-DONT-CARE*)
                        (yPos *GANTRY-DONT-CARE*)
                        (zPos *GANTRY-DONT-CARE*)
                        (xRot *GANTRY-DONT-CARE*)
                        (yRot *GANTRY-DONT-CARE*)
                        (zRot *GANTRY-DONT-CARE*) )
"Command moves the gantry to achieve the desired wrist endpoint position and
orientation in Cartesian space (does the required inverse kinematics).
Return codes:
-1:
-3: Error in solution for inverse kinematics"
```

5.5.1.2 CLOS Objects and Structures

```
(defclass GANTRY ()
    ((module :initarg :module :accessor module :initform nil
        :documentation "TCX module used to communicate with the gantry controller.")
))


(defclass GantryJointPosition ()
    ((xPos :initarg :xPos :accessor xPos :initform 0
        :documentation "The X position of the gantry cart (in millimeters).")
     (yPos :initarg :yPos :accessor yPos :initform 0
        :documentation "The Y position of the gantry cart (in millimeters).")
     (elbow-angle :initarg :elbow-angle :accessor elbow-angle :initform 0
        :documentation "Radian angle position of the gantry arm's elbow.")
     (WristRoll-angle :initarg :WristRoll-angle :accessor WristRoll-angle :initform 0
        :documentation "Radian angle position of the gantry arm's wrist roll")
     (WristPitch-angle :initarg :WristPitch-angle :accessor WristPitch-angle :initform 0
        :documentation "Radian angle position of the gantry arm's wrist pitch")
     (ToolRoll-angle :initarg :ToolRoll-angle :accessor ToolRoll-angle :initform 0
        :documentation "Radian angle position of the gantry arm's tool roll")
     (gripper-X :initarg :gripper-X :accessor gripper-X :initform 0
        :documentation "Opening span of the gantry arm's parallel gripper (in millimeters)")
))


(defclass GantryCartPosition ()
    ((xPos :initarg :xPos :accessor xPos :initform 0.0
        :documentation "The X position of the wrist endpoint position (in millimeters).")
     (yPos :initarg :yPos :accessor yPos :initform 0.0
        :documentation "The Y position of the wrist endpoint position (in millimeters).")
     (zPos :initarg :zPos :accessor zPos :initform 0.0
        :documentation "The Y position of the wrist endpoint position (in millimeters).")
     (xRot :initarg :xRot :accessor xRot :initform 0.0
        :documentation "The X rotational component of the wrist endpoint (in radians).")
     (yRot :initarg :yRot :accessor yRot :initform 0.0
        :documentation "The Y rotational component of the wrist endpoint (in radians).")
     (zRot :initarg :zRot :accessor zRot :initform 0.0
        :documentation "The Y rotational component of the wrist endpoint (in radians).")
))
```

5.5.2 Stereomotion Vision System API

5.5.2.1 *Methods and Functions*

```
(in-package :tcx)
(export '(
        PRISM                          ; function to get PRISM CLOS object
        PRISM-INIT                     ; initializes the PRISM system
        RESET                          ; resets the state of the PRISM system
        SHUTDOWN                       ; shuts down the PRISMSERVER status and releases it

        GET-JOINT-POSITIONS            ; query returns PRISM head status and configuration
        SET-JOINT-POSITIONS            ; command to change the PRISM head parameters

        MEASURE-DEPTHS                 ; measures a grid of depth points

        MEASURE-FLOWS                  ; measure a grid of real image flow vectors

        COPY                           ; used to copy tracker data objects
))
```

```
(defun PRISM-INIT (&key (timeout-in-secs 10)
                        (PRISM-name "prismcontrol") )
        "This function initializes TCX communication with the
        PRISM system, initializes PRISM subsystems, and it returns a PRISM CLOS object."
```

```
(defun PRISM ()
        "This function returns the current PRISM CLOS object."
```

```
(defmethod RESET ((state prism)
                        &key (timeout-in-msec 5000))
        "Initializes the PRISM boards and starts the head servos. Blocks until
        reset successfully completed."
```

```
(defmethod SHUTDOWN ((state prism)
                        &key (timeout-in-msec 10000))
        "Releases the PRISM system and head servos so that someone else can
        use it without exiting so you can make another call to RESET later.
        Blocks until reset successfully completed."
```

```
(defmethod GET-JOINT-POSITIONS ((state prism)
                        &key (timeout-in-msec 10000))
        "Query returns the absolute position of the PRISM head (joint angles are
        specified in radians) in the HeadPosition CLOS object."
```

```
(defmethod SET-JOINT-POSITIONS
                    ((state prism)
                    &key (await-completion t)
                          (timeout-in-msec 10000)
                          (panPos *PRISM-DONT-CARE*)
                          (tiltPos *PRISM-DONT-CARE*)
                          (vergePos *PRISM-DONT-CARE*) )
    "Command sets the absolute positions of the head axes (in radians)."

(defmethod MEASURE-DEPTHS
                    ((state prism) (pan float) (tilt float)
                    &key (verge *PRISM-DONT-CARE*)
                          (allow-head-motion t)
                          (perform-verge-scan t)
                          (display nil)
                          (Xinc 30)
                          (Yinc 30)
                          (Xdim 10)
                          (Ydim 10)
                          (Xorigin 50)
                          (Yorigin 50)
                          (timeout-in-msec 10000)
                          (parameter-check t)
                          (depth-array (make-array (list Xdim Ydim))) )
    "Instructs PRISM to measure distances to objects intersecting the pan-tilt gazepoint
```
directions. Argument descriptions:

PAN/TILT/VERGE is the global polar coordinate referenced to image center (in radians). (0,0,0) looks straight ahead.

ALLOW-HEAD-MOTION when t, the PRISM system will (if required) pan and tilt the stereo head to bring specified measurement points into the field of view (FOV).

XINC/YINC is the delta factor used to measure depth points (e.g., every fourth Y (row) pixel)

XDIM/YDIM is the number of depth measurements in a given axis (e.g., 10 row depths measures per column)

XORIGIN/YORIGIN is the upper left in the image(row,col) coords of the depth measurement array

TIMEOUT-IN-MSEC specifies how long PRISM may be nonresponsive without generating an error. "

5.5.2.2 CLOS Objects and Structures

```
(defclass PRISM ()
          ((module :initarg :module :accessor module :initform nil
                :documentation "TCX module used to communicate with the PRISM controller.")
))
```

```
(defclass ImageData ()
        ((row :initarg :row :accessor row :initform 0.0
            :documentation "The row position in the image frame.")
        (col :initarg :col :accessor col :initform 0.0
            :documentation "The col position in the image frame.")
        (rowVel :initarg :rowVel :accessor rowVel :initform 0.0
            :documentation "Row image velocity at the current position in the image frame.")
        (colVel :initarg :colVel :accessor colVel :initform 0.0
            :documentation "Col image velocity at the current position in the image frame.")
))


(defclass HeadJointPositions ()
        ((panPos :initarg :panPos :accessor panPos :initform 0.0
            :documentation "The head pan position (in radians).")
        (tiltPos :initarg :tiltPos :accessor tiltPos :initform 0.0
            :documentation "The head tilt position (in radians).")
        (vergePos :initarg :vergePos :accessor vergePos :initform 0.0
            :documentation "The head verge position (in radians).")
))
```

```
(defclass HeadPosition (HeadJointPositions)
        ((panMin :initarg :panMin :accessor panMin :initform 0.0
            :documentation "The leftmost pan field-of-view (in radians).")
        (panMax :initarg :panMax :accessor panMax :initform 0.0
            :documentation "The rightmost pan field-of-view (in radians).")
        ; static position bounds
        (tiltMin :initarg :tiltMin :accessor tiltMin :initform 0.0
            :documentation "The lower tilt field-of-view (in radians).")
        (tiltMax :initarg :tiltMax :accessor tiltMax :initform 0.0
            :documentation "The upper tilt field-of-view (in radians).")
        (vergeMin :initarg :vergeMin :accessor vergeMin :initform 0.0
            :documentation "The closest verge position (in radians).")
        (vergeMax :initarg :vergeMax :accessor vergeMax :initform 0.0
            :documentation "The largest verge position (in radians).")
        ; dynamic bounds
        (panMaxVel :initarg :panMaxVel :accessor panMaxVel :initform 0.0
            :documentation "The maximum pan axis velocity (in radians/second).")
        (tiltMaxVel :initarg :tiltMaxVel :accessor tiltMaxVel :initform 0.0
            :documentation "The maximum tilt axis velocity (in radians/second).")
        (vergeMaxVel :initarg :vergeMaxVel :accessor vergeMaxVel :initform 0.0
            :documentation "The maximum verge axis velocity (in radians/second).")
        ; head calibration parms
        (pixel_x_size :initarg :pixel_x_size :accessor pixel_x_size :initform 0.0
            :documentation "The PRISM pixel size in angle subtended (in radians).")
        (pixel_y_size :initarg :pixel_y_size :accessor pixel_y_size :initform 0.0
            :documentation "The PRISM pixel size in angle subtended (in radians).")
        ; PRISM mask parms
        (filter_w :initarg :filter_w :accessor filter_w :initform 0.0
            :documentation "The PRISM filter size.")
        (cor_wind_xdim :initarg :cor_wind_xdim :accessor cor_wind_xdim :initform 0.0
            :documentation "The PRISM correlation window dimension in X.")
        (cor_wind_ydim :initarg :cor_wind_ydim :accessor cor_wind_ydim :initform 0.0
            :documentation "The PRISM correlation window dimension in Y.")
        (cor_wind_xpos :initarg :cor_wind_xpos :accessor cor_wind_xpos :initform 0.0
            :documentation "The PRISM correlation window center in X.")
        (cor_wind_ypos :initarg :cor_wind_ypos :accessor cor_wind_ypos :initform 0.0
            :documentation "The PRISM correlation window center in Y.")
))


(defclass Measured-Depths (HeadVelocity)
        ((range-data :initarg :range-data :accessor range-data
            :initform (make-array '(10 10) :element-type 'float :initial-element 0.0)
            :documentation "The range data for the tracked object position (in feet).")
        (start-head-position :initarg :start-head-position :accessor start-head-position
            :initform (make-instance 'HeadJointPositions)
            :documentation "The head position when the range data measurement began
                        (a HeadJointPositions CLOS object).")
        (end-head-position :initarg :end-head-position :accessor end-head-position
            :initform (make-instance 'HeadJointPositions)
            :documentation "The head position when the range data measurement finished
                        (a HeadJointPositions CLOS object).")
))
```

```
(defclass Measured-Flows (HeadVelocity)
        ((Xmotion :initarg :Xmotion :accessor Xmotion
             :initform (make-array '(10 10) :element-type 'float :initial-element 0.0)
             :documentation "The real X flow data (in radians/sec).")
        (Ymotion :initarg :Ymotion :accessor Ymotion
             :initform (make-array '(10 10) :element-type 'float :initial-element 0.0)
             :documentation "The real X flow data (in radians/sec).")
        (confidences :initarg :confidences :accessor confidences
             :initform (make-array '(10 10) :element-type 'integer :initial-element 0)
             :documentation "The flow data confidences (0 for none, 1 for OK).")
        (start-head-position :initarg :start-head-position :accessor start-head-position
             :initform (make-instance 'HeadJointPositions)
             :documentation "The head position when the range data measurement began
                            (a HeadJointPositions CLOS object).")
        (end-head-position :initarg :end-head-position :accessor end-head-position
             :initform (make-instance 'HeadJointPositions)
             :documentation "The head position when the range data measurement finished
                            (a HeadJointPositions CLOS object).")

        ))
```

5.5.3 Color Perception API

5.5.3.1 *Methods and Functions*

```
(in-package :user)
(export '(
        COLOR-INIT
        COLOR
        COLOR-LEARN-OBJECT
        COLOR-FIND-OBJECTS
))
```

```
(defun COLOR ()
```
   "Returns the CLOS/LISP color communications object."

```
(defun COLOR-INIT (&key (timeout-in-msecs 30000)
                        (color-name "colorserver")
                        (reset t) )
```
   "This function initializes TCX communication with the color vision system and it returns
   a CLOS/LISP color communications object."

```
(defun COLOR-LEARN-OBJECT (&key (timeout-in-msecs 5000))
```
   "This function has the color vision system learn an object that is the only color
   object within the field of view. It returns a model index for the learned object. "

```
(defmethod COLOR-FIND-OBJECTS (  (model-indices list)
                                 &key (timeout-in-msecs 5000)
                                      (module (module (color))))
```
"This function has the color vision system find an object for which a model has been learned. These learned models are passed in as a list of model indices returned by calls to COLOR-LEARN-OBJECT. It returns a list of the corresponding CLOS objects and their matches."

5.5.3.2 CLOS Objects and Structures

```
(defclass found-object (2D-object color-object)
     ((confidence :initarg :confidence :accessor confidence :initform nil
          :documentation "The object confidence from 0 (bad) up (unnormalized).")
))
```

```
(defclass 2D-object ()
     ((center :initarg :center :accessor center :initform nil
               :documentation "Object centroid in (<row> <col>).")
      (len :initarg :len :accessor len :initform nil
               :documentation "Object length (in pixels).")
      (width :initarg :width :accessor width :initform nil
               :documentation "Object width (in pixels).")
      (axis :initarg :axis :accessor axis :initform nil
               :documentation "Principal axis of object (in (<axis x> <axis y>) pixel coords).")
      (pixel-count :initarg :pixel-count :accessor pixel-count :initform nil
               :documentation "Number of pixels in object.")
))
```

```
(defclass color-object ()
     ((hue :initarg :hue :accessor hue :initform nil
               :documentation "The hue of the object parts.")
      (saturation :initarg :saturation :accessor saturation :initform nil
               :documentation "The saturation of the object parts.")
))
```

5.5.4 Sound API

5.5.4.1 Methods and Functions

```
(in-package :tcx)
(export '(
     PLAYER-INIT
     PLAY

     ))
```

```
(defmethod PLAYER-INIT ()
     "This method initializes the TCX sound player."
```

```
(defmethod PLAY ((playfiles string)
                   &key (await-completion t)
                        (timeout-in-msec 1000) )
```
"This method plays a list of sound files. For example, these sounds can be recorded using the soundtool utility."

5.5.4.2 CLOS Objects and Structures

The Sound API has no object or structure definitions.

5.5.5 Integrated Actions (Programmed Behaviors) API

The integrated actions (behaviors) described in this subsection sequence the atomic actions described in previous subsections to perform a higher level capability to the agent or agent designer.

5.5.5.1 Methods and Functions

```
(in-package :tcx)
(export '(
        FIND-CUSTOMER-HEAD      ; scans for a customer and if found scans to their head

        TRACKER-ON              ; turns the head/eye tracker on
        TRACKER-OFF             ; turns the tracker off
))
```

```
(defmethod FIND-CUSTOMER-HEAD
                (&key (display nil)
                        (verge-pos 0.16)
                        (customer-depth-range '(7.0 9.5))
                        (max-pixels-from-image-center  10) ; in pixels
                        (empty-top-row-thresh 2)
                        (depth-row-pixel-size 10)
                        (depth-col-pixel-size 10)
                        (tilt-head-seeking-move-in-pixels 25)
                        (prism (prism))
                        (gantry (gantry))
                        (do-verge nil)
                        (speedLimit .2)
                        (prism-scan-bounds '( (-.045 -.1) (.175 -.1))))
```
"This method scans the counter until a body (i.e., something large in range at waist height) is found. It then scans up looking for the head. If no one is found, it returns NIL. Otherwise, it returns '((pan-in-radians tilt-in-radians) (top-head-row center-head-col))."

```
(defmethod TRACKER-ON
                    ((state prism)
                    &key (row 256) (col 256)
                            (allow-head-motion        t)
                            (allow-verge-motion       t)
                            (allow-window-motion   t)
                            (timeout-in-msec          5000)
                            (update-rate-in-msec     1000)
                            (update-tcxID *TRACKER_STATUS_DATA_ID*)
                            (num-successive-bad-correlations-allowed 2)
                            (parameter-check t) )
```
"Turns the PRISM head tracker on. Argument descriptions:
  ROW & COL are the image coordinates the tracker should start at when ALLOW-HEAD-MOTION is non-nil.
  ALLOW-HEAD-MOTION determines whether the head is moved to maintain tracking.
  ALLOW-VERGE-MOTION determines whether the head vergence is adjusted during tracking.
  UPDATE-RATE-IN-MSEC specifies the rate in msec that measurments are returned. Granularity of this update rate is 1/30th second (a frame rate).
  UPDATE-tcxID specifies the TCX id the PRISM system will use to send back data every (UPDATE-RATE) per frame.
  NUM-SUCCESSIVE-BAD-CORRELATIONS-ALLOWED specifies the number of frames of successive bad correlations allowed before tracking is aborted.
  ALLOW-WINDOW-MOTION determines whether the tracker follows an image patch (t) or simply reports motion vectors at a fixed image position (nil).
Returns a function closure that when FUNCALLed with no arguments, it awaits and returns a measurement object (note that this measurement object is reused to save space, so you must copy fields you wish to retain before making another FUNCALL; e.g., (COPY <object>)). If you want to flush all past data to ensure the freshest measurement, then FUNCALL this function closure with the argument T (e.g.., (FUNCALL (TRACKER-ON (PRISM)) T)). Thus funcall returns one of the following values: <TRACKER-DATA-CLOS-OBJECT> if the tracker is running and a measurement has been returned :STEREO-CORRELATION-LOSS if the stereo system lost correlation, :MOTION-CORRELATION-LOSS if the motion system lost correlation, and :TRACKER-SHUTTING-DOWN is either has lost correlation for more than NUM-SUCCESSIVE-BAD-CORRELATIONS-ALLOWED."

```
(defmethod TRACKER-OFF ((state prism)
                               &key (timeout-in-msec  5000) )
```
"Turns the PRISM head tracker off."

5.5.5.2 CLOS Objects and Structures

In addition to the objects and structures defined in the previous sections, the following are defined:

```
(defclass TrackerData (HeadJointPositions ImageData HeadVelocity)
        ((range :initarg :range :accessor range :initform 0.0
            :documentation "The range for the tracked object position measured in feet.")
        (mean-range :initarg :mean-range :accessor mean-range :initform 0.0
            :documentation "The mean-range for the tracked object position in feet.")
        (mean_pan_motion :initarg :mean_pan_motion :accessor mean_pan_motion :initform 0.0
            :documentation "The mean_pan_motion for the tracked object position in feet.
                            Approx equal to image_motion_xvel + head_motion_xvel.")
        (mean_tilt_motion :initarg :mean_tilt_motion :accessor mean_tilt_motion :initform 0.0
            :documentation "The mean_tilt_motion for the tracked object position in feet.
                            Approx equal to image_motion_xvel + head_motion_xvel.")
))
```

6 PC-Serbot TESTBED IMPLEMENTATION

This section discusses the PC-Serbot implementation of the testbed Application Programmer's Interface (API) that was described in Section 5. This implementation was developed and used to evaluate agent performance (as will be described in Section 7).

The testbed API was specifically designed to allow realization as either a software simulation or using actual robot and vision hardware. In considering which implementation, or both, would be pursued, we considered properties of simulation and actual hardware realization.

Software simulation of the testbed API has the following advantages:

*   Controllability of testbed parameters provides high repeatability of results.
*   Software simulation simplifies usage by a wide research and development community.

Disadvantages of software simulation:

*   Simulation of perception, timing, mechanics and other effects is limited in its realism and fidelity.
*   Because simulations need not faithfully model realistic conditions, and simulation assumptions and implementation can often induce subtle performance effects, simulation results can be suspect.
*   Usefulness of a simulation may be largely limited to an analysis function without necessarily leading to physical realizations or demonstrations in useful physical problem domains.

Hardware realization and physical demonstration of the testbed API has the following advantages:

*   Testbed operations and performance reflect achievable capabilities.
*   If an agent can physically perform a task, then performance assessments become more believable than in a simulation.
*   Interesting and important agent and testbed construction principles are explored in a physical realization that may not be especially relevant in a simulation (e.g., interoperability, parallel resource management, failure detection and recovery).
*   The integration of actual physical capabilities to perform useful tasks is itself an important element of work on reactive agents.

Disadvantages of hardware realization:

*   The flip side of added realism is that the physical world and hardware generally require substantial time and effort to engineer and integrate (though this is getting simpler as more commercial off-the-shelf components become available).
*   Hardware realization adds substantial component costs.
*   Physical systems are prone to variability that is hard to control and model.

The Teleos testbed is a physical implementation of the testbed API. The physical implementation provided a more realistic and useful environment in which to test and evaluate reactive agents, and physical testbeds have been rarer among researchers doing work on agent evaluation. Though a physical realization is a far more complex undertaking than a simulation,

Teleos' expertise in real-time perception and robotics provided substantial technical and equipment resources needed to implement the testbed API in actual hardware.

Section 6.1 describes the hardware testbed components that were developed to support agent execution of the benchmark tasks. Section 6.2 discusses how software integration was achieved in a heterogeneous and distributed computing/hardware environment. Section 6.3 then discusses software and algorithmic bases used to implement the testbed API capabilities.

6.1 Hardware Testbed Components

Figure 7 shows a photo of the robot hardware components used to realize the PC-Serbot testbed API. Two monochrome stereo cameras mounted in a stereo verge unit are used to support real-time stereo, visual motion and visual tracking capabilities. In addition, a color camera is provided for color object detection. Both the stereo and color cameras are mounted on a pan-tilt unit that allows for control of the camera viewing direction. The cameras and active head are mounted on an upper torso gantry robot that performs X-Y translation on a plane parallel to the package stocking areas and customer service counter. Motor power drivers mounted on this torso interface with the host computers to provide power and control to the active head and gantry DC servomotors. Mounted beneath the gantry torso is a four degree of freedom (DOF) arm (elbow,
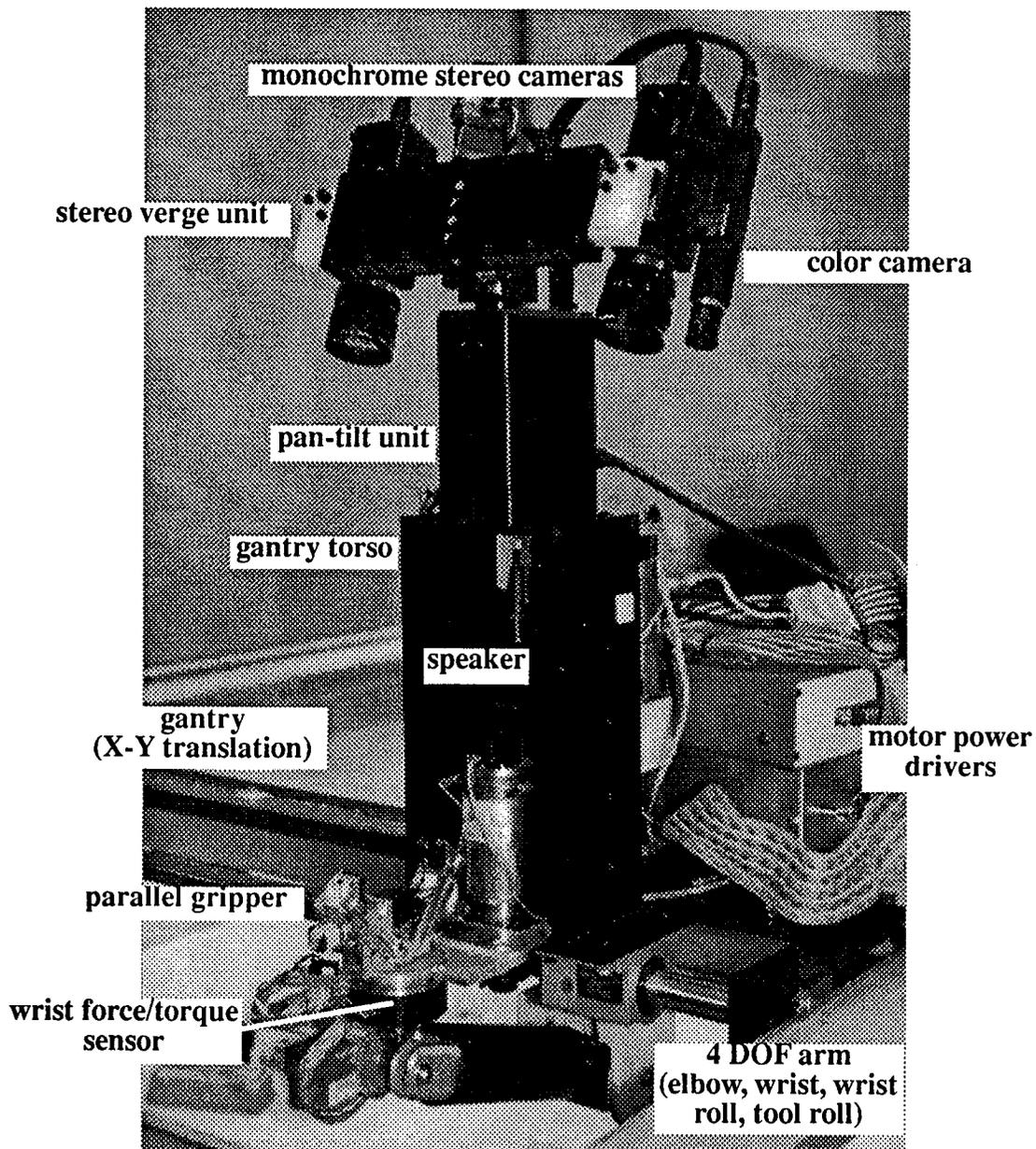

Figure 7: PC-Serbot Robot Hardware Components

wrist, wrist roll and tool roll). A parallel gripper mounted to the arm provides end effector grasping capabilities. A wrist force/torque sensor (6 DOF) between the gripper and arm provides the ability to determine effector/object contact and effector dynamics. A speaker mounted within the gantry torso provides for audio output.

Figure 8 shows the PC-Serbot system architecture. The robot hardware components shown in Figure 7 are controlled by distributed computers that are networked together by Ethernet. The PRISM-3 system is a stand-alone computing system which tightly integrates the stereo cameras and active head to perceive depth and visual motion, and to perform head control and visual tracking. The color vision system integrates the color camera with a high speed video digitizer and computing on the host Sparcstation to implement the color recognition capability. A gantry/ arm/gripper controller (henceforth, the "gantry controller") integrates these robotic components with motor controllers and servos, control software and computing on an IBM PC compatible computer. The sound player drives the gantry speaker using audio capabilities on a Sun 2+. As shown in Figure 8, the testbed API defines the interface to these actions and resources.

As shown in Figure 7, the gantry robot X-Y translation stage and torso is comprised of two orthogonal rail and pillow-block bearing systems moved by independent DC servomotors with optical encoder feedback. A four degree of freedom (DOF) arm (elbow, wrist, wrist roll and tool roll) was inverted and mounted beneath the gantry torso to provide access to the working surface and the 7" of space above it. A parallel gripper powered by a DC servomotor with optical encoder
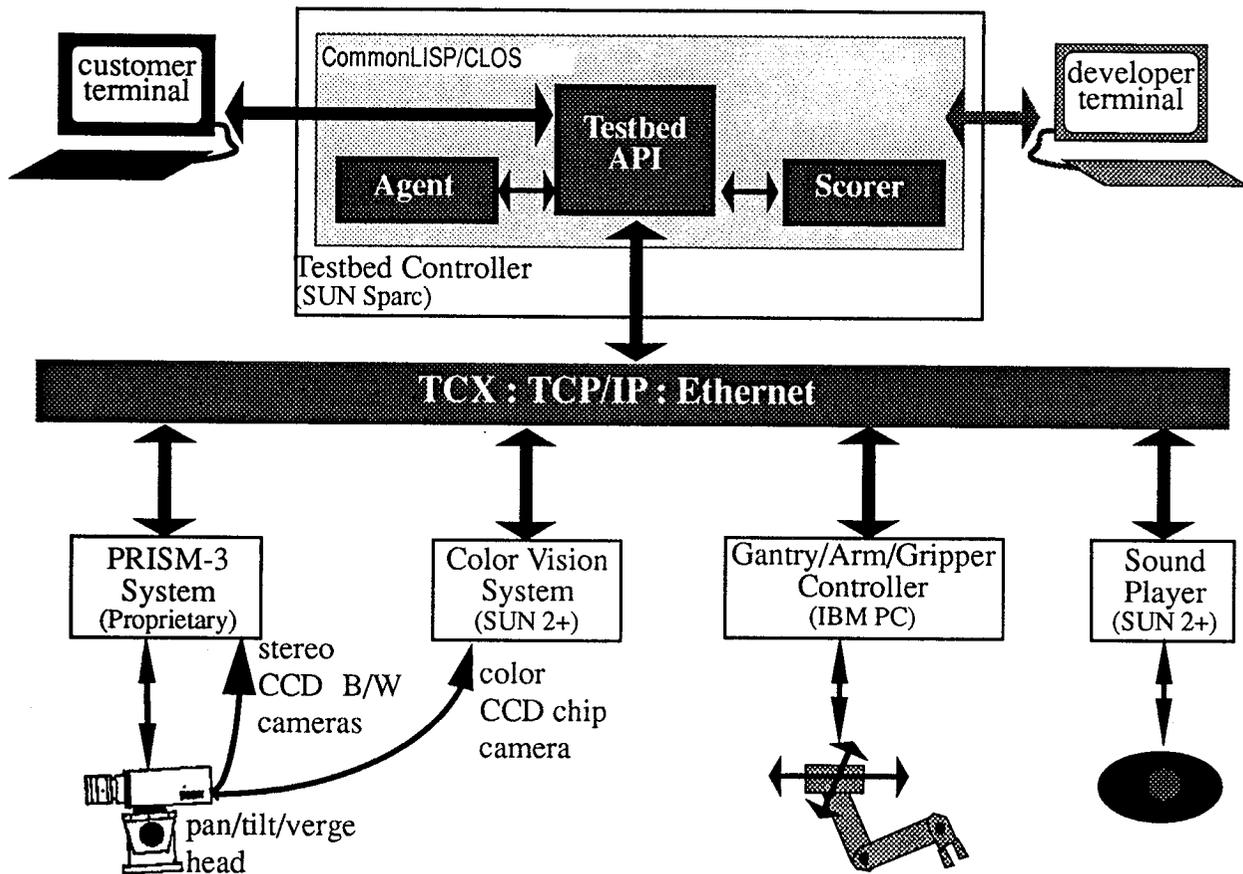

Figure 8: PC-Serbot System Architecture

feedback developed by Teleos provides an end effector with a maximum gripper opening of about 2-3/4". The parallel gripper is mounted to a 6 DOF wrist force/torque sensor $[(\dot{x}, \dot{y}, \dot{z}), (\ddot{x}, \ddot{y}, \ddot{z})]$ that provides force and torque measurements at the end effector. The gripper and force/torque sensor are mounted at the tip of the robot arm. The seven DC servomotors in the robot (four motors in the arm, one motor in the gripper, and two motors in the gantry translation stage), their integral optical encoders, and the force/torque sensor output are connected to the gantry PC computer via a single EISA board which provides basic PID motor control signals and quadrature encoder support. The PC-based gantry controller software developed by Teleos performs servocontrol and support functions, and it provides Ethernet/TCX interfaces to the testbed API.

The PRISM-3 stereo-motion system was developed by Teleos for frame-rate measurement of stereoscopic depth and visual motion. This third generation system is built on a stand-alone VME chassis which houses two custom boards developed by Teleos, a convolver and correlator, in addition to a high speed frame grabber, embedded 68040 computer and the active head motor controller board. A detailed description of the active head is provided in Section 6.3.2.3. The PRISM-3 system runs OS9 and it has Ethernet and TCP/IP network connections. A more detailed description of PRISM-3 hardware and principles of operation is provided in Section 6.3.2 and [50].

The color vision system was developed by Teleos for near frame-rate detection and description of packages within the testbed environment. A Sony XC-999 color CCD chip camera provided good quality imagery within a miniature package that was directly mounted on the active head. A Data Cell S2200 color digitizer was used since its S-bus memory-mapped I/O provides for very fast image data transfer. In addition, this digitizer supports 1/2 and 1/4 image downsampling, look-up table value mapping, and fast NTSC graphics output support. The color feature and region extraction, and object model matching is performed in software (in CommonLISP/CLOS). A more detailed description of the color vision system is provided in Section 6.3.3.

The sound player system provides an Ethernet/TCX interface to the testbed API for playing pre-recorded sound files on the gantry speaker through an audio amplifier. The SunOS SoundToolkit was used to execute this capability on Sun Sparc workstations.

As shown in Figure 8, the testbed API, the embedded agent, and the agent scorer (which calculates performance metrics) operate within an Allegro CommonLISP/CLOS environment running on the testbed controller machine (a SUN Sparc). All agent interaction with the testbed environment occurs through the testbed agent API which was detailed in Section 5. The scorer maintains information about testbed API calls to compute agent performance metrics as detailed in Section 6. As noted earlier, a customer terminal provides for testbed customer API support as described in Section 5.4. In order to support testbed and agent developers, as shown in Figure 8, a developer terminal is provided for direct interaction with the CommonLISP environment.

6.2 Software Integration of Distributed Heterogeneous Computing/Hardware

Robotic and other complex physical systems commonly require integration across heterogeneous computers and other equipment. Often, there can be a need for relatively high data bandwidth between computing systems. In addition to variations across hardware and equipment,

such robotic systems may use more than one computer programming language and methodology (e.g., C/C++, CommonLISP/CLOS). An important element of early implementation efforts was the determination of a hardware and software architecture and components that provided for

- multiple heterogeneous and distributed computers and computing systems
- the use of an interface protocol that allowed the use of multiple computer languages
- relatively high data bandwidth between distributed computers to provide for real-time operations
- straightforward and simple development of interfaces
- cost effective and straightforward technical achievability of interoperability.

Ethernet and TCP/IP were selected to provide distributed connectivity since they have good data bandwidth (e.g., 10 Mbits/sec), they are available across a broad range of computing equipment and operating systems, and a wide range of software allows straightforward and cost effective usage.

In order to provide for programming of interprocess communication and data transfer, the TCX software programming package developed at CMU was used. TCX provides a C-language program interface for point-to-point communications that is supported on several computer and operating systems platforms. TCX is currently being used for integration of the ARPA Unmanned Ground Vehicle Program. A more detailed description of TCX is provided in Appendix B. To support the testbed API, which is based upon CommonLISP/CLOS, a TCX-CL/CLOS interface was developed on top of the basic TCX package. The next subsection overviews the layers of software built to provide transparent interoperability between the agent processes and distributed and heterogeneous subsystem components (e.g., gantry robot, vision systems).

6.2.1 Distributed System Software Architecture

Several layers of software and network support provide for transparent distributed interoperability between the agent processes and other subsystem components. Figure 9 shows the PC-Serbot distributed system software architecture. An agent interfaces and interacts with testbed subsystems via the CL/CLOS testbed API defined in Section 5. In turn, the testbed API must interface with these other heterogeneous subsystems.

In order to provide interprocess communication in CL/CLOS, a TCX-CL/CLOS interface (API) was built to allow direct interprocess communication between distributed CL processes. This interface was built using the CommonLISP/CLOS foreign function interfaces provided in Allegro CommonLISP to call the C routines that make up the basic TCX package. The TCX software programming package developed at CMU provides for interprocess transfer of data structures while compensating for data format differences between varying machine and OS architectures. The TCX-CL/CLOS API is analogous to the C TCX package, though some additional error and parameter checking was added. In addition, support was provided to allow concurrent CommonLISP lightweight processes to use the API without harming the TCX context or communications state. TCP/IP is the underlying protocol used by TCX to eventually communicate over a physical Ethernet network connection to other subsystems and processes. TCX-CL/CLOS function calls and data structures are defined in Section 6.2.2.

As shown in Figure 9, subsystems interface with the testbed API via a TCX interface defined for them. For example, *Subsystem1* executing on *machine1* may be the gantry controller (written in C) which provides TCX interfaces for moving the robot, querying its position, etc. Alternatively, *Subsystem n* executing on *machine n* may be an integrated action (behavior) which is written in CommonLISP/CLOS. Integration of a new subsystem capability only requires that a TCX interface be defined for it and made available to the testbed API via the TCX-CL/CLOS package.

Because TCX and TCX-CL/CLOS provide for process-to-process interface descriptions, it overcomes the requirement that agents and supporting subsystems use the same language, operating system, computer hardware, or processing model. This key advantage makes it far simpler to integrate capabilities that may have been developed separately or for which alternative implementation approaches are more feasible. As a result, more capabilities can often be brought to bear upon an agent task without the requirement for extensive and expensive reengineering.
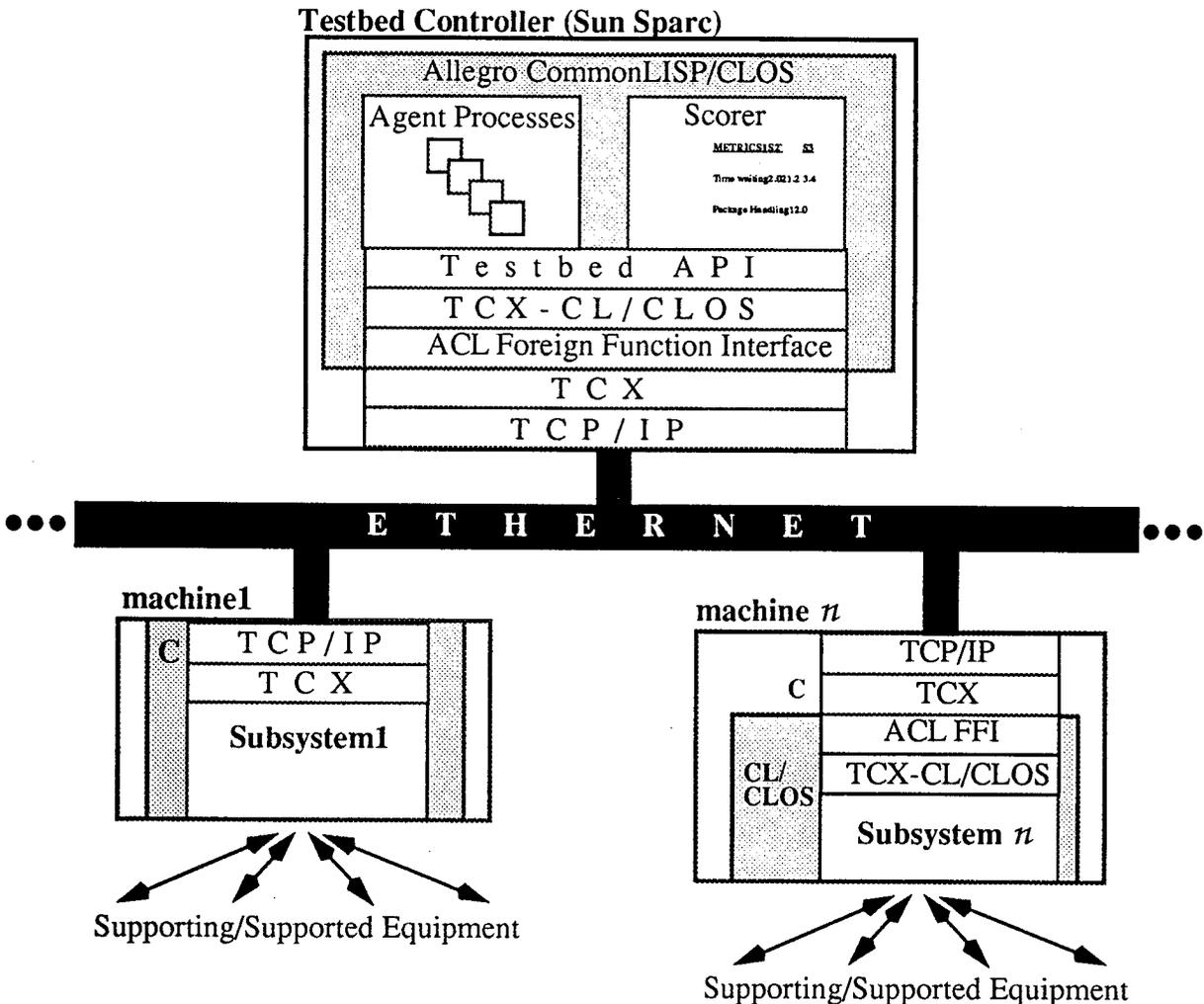

Figure 9: PC-Serbot Distributed System Software Architecture

These benefits are key reasons why TCX was used for this effort and why it has been adopted by the ARPA UGV program.

6.2.2 TCX-CL/CLOS: CommonLISP/CLOS API to TCX

```lisp
(in-package :tcx)
(export '(
    *tcxNIL*
    tcxINIT
    tcxCONNECT
    tcxSEND
    tcxRECEIVE
    tcxFLUSH
    tcxPARSER
    tcxDECODE
    tcxFREE
))
```

```lisp
(defmethod tcxINIT ((moduleName string)
                &optional (timeout-in-msec 10000))
    "This method initializes the module name with the TCX server. EXAMPLE USAGE:
(tcxINIT "PRISM")."
```

```lisp
(defmethod tcxCONNECT ((moduleName string)
                &optional (timeout-in-msec 10000) )
    "This method connects the module name with the TCX server. EXAMPLE USAGE:
(setf prism_module (tcxCONNECT "PRISM"))"
```

```lisp
(defmethod tcxSEND ((module_struct integer)
                    (id integer)
                    (data_buffer t)
                    (length integer)
                    (fmt_struct integer)
                &optional (timeout-in-msec 10000) )
    "Send information to a module. Returns status.
    EXAMPLE USAGE: (tcxSEND prism_module 13 'EL(32,100)' 10 0).
    Argument description:
    (module_struct integer)     ; the object returned from a call to tcxCONNECT
    (id integer)                ; a unique message identifier (a.k.a. mailbox number)
    (data_buffer t)             ; the data to be sent
    (length integer)            ; if data_buffer is a byte buffer, the number of bytes
    (fmt_struct integer)        ; if data_buffer is a struct, its declaration (see tcxFORMAT)"
```

```
(defmethod tcxRECEIVE ((module_struct integer)
                              &optional (id *tcxNIL*)
                                       (data_buffer (tcxBUFFER 200))
                                       (fmt_struct *tcxNIL*)
                                       (timeout-in-msec 10000) )
    "Receive information from a module. Returns '(<result code> <received data> <tcxID>).
    EXAMPLE USAGE: (tcxRECEIVE prism_module 13 in-struct
                              (tcxPARSER '{int int long}') 10)
```
　　Argument description:
　　　(module_struct integer)　　　　; the sender object returned from a call to tcxCONNECT
　　　(id integer)　　　　　　　　　; a unique message identifier (a.k.a. mailbox number)
　　　data_buffer　　　　　　　　　; the data buffer that is to receive incoming data
　　　(fmt_struct integer)　　　　　; if data_buffer is a struct,, its declaration (see tcxFORMAT)
　　　(timeout-in-msec integer)　　　; how long before a RECEIVE fails when nothing is there
　　　　　　　　　　　　　　　　　　; (min. timing, can't be assured of exact timeout period)."


```
(defmethod tcxFLUSH ((module_struct integer)(id integer)
                              &optional (timeout-in-msec 10000) )
    "Flushpending items from tcxRECEIVE.
    EXAMPLE USAGE: (tcxFLUSH prism_module 13) .
```
　　Argument description:
　　　(module_struct integer)　　　　; the object returned from a call to tcxCONNECT
　　　(id integer)　　　　　　　　　; a unique message identifier (a.k.a. mailbox number)"


```
(defmethod tcxPARSER ((parse_format string)
                              &optional (timeout-in-msec 10000) )
    "This method returns a parser structure pointer passed to the tcxSEND, tcxRECEIVE,
    and tcxDECODE commands. Parse format strongs are of the form: {<list of scalar
    types>} where scalar types are: int, long, float, double, byte, char, short, ... (see TCX/
    TCA manual for additional supported types)."
```


```
(defmethod tcxDECODE ((fmt_struct integer)
                              (format_buffer string)
                              (byte_buffer string)
                              &optional (timeout-in-msec 10000) )
    "This method takes a byte_buffer and a format description and places the formatted data
    into the format_buffer."
```


```
(defmethod tcxFREE ((format_struct integer) byte_buffer
                              &optional (timeout-in-msec 10000) )
    "This method frees a TCX byte_buffer using a TCX format spec."
```

6.3 Testbed System Components

This section describes in more detail the implementation used to achieve the testbed API capabilities defined in Section 5.

6.3.1 Gantry Controller

The hardware for the gantry/arm/gripper controller (a.k.a. the gantry controller) was described in Section 6.1. The gantry controller software is written in C and runs on a 386/87 machine. The TCX interface supported by the gantry controller is described in Section 5.5.1.

The gantry controller provides TCX interfaces to specify gantry position by both joint position and end effector position. Joint position is set by direct control of motor position using encoder feedback and linkage relations. End effector position (i.e., gripper wrist) may be specified with 6 parameters (translation and rotation in 3D), and the gantry controller computes the inverse kinematics and required linkage positions. The motor driver board within the gantry computer contains HP HCTL-1000 DC servomotor control chips which take in a quadrature (optical encoder) signal, PID control parameter settings, and desired position to appropriately move the gantry linkages. The gantry controller software programs the HCTL-1000 chips to achieve desired linkage position and motion dynamics.

The gantry controller also supports force-feedback operations. The gantry controller provides for termination of an arm movement command when an object is contacted. Object contact is determined using the wrist force/torque sensor, and it is detected as a vector spike in the force/torque measured over time. The magnitude of this spike may be invocationally specified, and its default value is a wrist force greater than the forces exerted by movement of the arm or gripper (i.e., a wrist force that must have been caused by contact with another object).

A command is provided for closing the gripper until an object is grasped. Grasp detection is performed by noting when the gripper DC servomotor stalls (as detected by its integral optical encoder, not by a motor overcurrent condition). The resulting gripper position may then be noted to determine if an object is in the gripper, and if so, its grasped dimension. The gripper is limited to a maximum opening of 2-3/4", and practically, an object axis greater than about 80% of this opening may be difficult to grasp reliably.

6.3.2 PRISM-3 System

The Prism-3 stereo system described in [50], employs the sign-correlation algorithm for stereo matching [53], in a demand driven accelerator architecture that allows a hundred or so directed range measurements to be made within a 30 millisecond camera frame time.

One of the principal advantages of this demand directed approach is that it gives task directed programs close control over sensor resource allocation and it makes it easier to return more complete information about what was measured to be returned (e.g., range, height, sharpness or bimodality of correlation peak).

The demand-driven approach allows applications to accomplish the task at hand with increased efficiently and precision. For example, range measurements can be made just at the locations required to accomplish a perceptual task such as discerning an occluding edge. In this example, the reduced measurement bandwidth makes it easier to deliver more data such as

information about the shape of the stereo cross-correlation peak that might aide the application process to resolve the edge position.

The demand-driven methodology simplifies communication between higher and lower level perceptual processes. In many instances, intuitive top-down control can be achieved by simply pointing the device and setting the size of its measurement field (zoom). At the same time, we want the lower-level measurement facility to return a richer result than local range and confidence. For example, a representation of the stereo cross-correlation peak shape with sufficient fidelity to recover height, width and possible bimodality allows the application developer to deploy the generic stereo measurement resources effectively to accomplish specific perceptual tasks.

6.3.2.1 Sign-Correlation Algorithm

The sign-correlation algorithm is an area correlation approach applied to binarized Laplacian-of-Gaussian ($\nabla^2 G$) filtered images. The main processing steps for this algorithm are as follows:

3. Input stereo images are bandpass filtered by $\nabla^2 G$ convolution with a typical kernel center diameter, w, of 8 pixels (20 pixel overall diameter).

4. Convolved images are binarized by hard clipping (effectively taking the sign bit). This gives a sign representation that captures the zero-crossing primitives of Marr and Poggio [45].

5. Image similarity is measured by standard sum of products correlation on image windows with typical diameters of 32 pixels. This is an important difference from the Marr-Poggio-Grimson zero-crossing contour matching approach which has a serious noise sensitivity problem. The sign-correlation approach in contrast is extremely noise tolerant [52].

6. Disparity is measured by finding the peak correlation. Disparity search range is dynamically adjusted to the needs of the application. Typically correlation measurements are sampled at intervals of $w/4$ to decrease the number of correlation checks required to cover a given disparity search range. This relies the width of the correlation peak which is on the order of w, the convolution filter center diameter.

7. Confidence is estimated from an analysis of peak height, shape, and uniqueness. The measured correlation values are automatically normalized since the sign images are binary.

8. Subpixel disparity resolution is obtained by fitting a quadratic function to three correlation measurements at neighboring integer disparities about the peak position.

9. Individual disparity measurements are transformed into range coordinates and passed back to the calling application.

6.3.2.2 PRISM-3 Architecture

The sign correlation algorithm has been implemented on the system illustrated in Figure 10. The stereo cameras are mounted on a pan-tilt-vergence mechanism illustrated in Figure 8. The baseline of the cameras is fixed at 22.2 cm., and the range of motion for each camera is from parallel to 10 degrees. This corresponds to a minimum target distance of about 60 cm. The head can move through a 180 degree rotation in under a second and exhibits a positioning repeatability on the order of 50 arc seconds standard deviation in pan, 20 arc seconds in tilt, and 6 arc seconds in arc seconds in vergence.

The two video cameras are operated in a mode where they share the same pixel clock in order to minimize timing skew between the cameras that would result from only using horizontal and vertical video synchronization signals. The left and right camera video is digitized using commercial (DataCube) digitizer hardware and parallel digital video streams are fed to two dedicated Laplacian-of-Gaussian convolvers (developed by Teleos). These convolvers allow video rate convolution with operator center diameters ranging from 1.6 pixels to 16.6 pixels.

Digital convolution video signals are fed from the two convolvers to a binary correlator board (also developed at Teleos) which carries out high speed correlations on the sign bits of the input video streams.

The Prism-3 correlator board performs 36 correlations in parallel on rectangular windows of adjustable size. The correlator board is operated by an external control processor (currently a 68040 single board computer). At the start of a measurement cycle, this processor writes the pixel coordinates of the next measurement to be made into registers on the correlator along with information about the disparities at which correlation measurements are to be made. A set of correlations with 32 by 32 pixel windows at 36 different disparities takes 100 microseconds to complete. The correlation results are then read into the control processor. If a well formed peak is identified in the data, quadratic interpolation is used to refine the peak disparity. These steps on the CPU take an additional 200 microseconds.

With correlations taken at even pixel disparities at a single vertical disparity, the above 300 microsecond cycle allows a disparity peak to be located in a 72 pixel disparity search range with a third to a tenth of a pixel resolution. Vertical disparity errors between 1 and 2 pixels are well tolerated.

Organizing the system to take a single range measurement at a time makes it easier to write higher level procedures that:

1. Adjust vertical disparity as a function of image position and disparity. Coarse models of epipolar line positions are easy to calibrate and allow effective operation in the presence of severe image rotation, vertical misalignment, trapezoidal and pin cushion distortion.
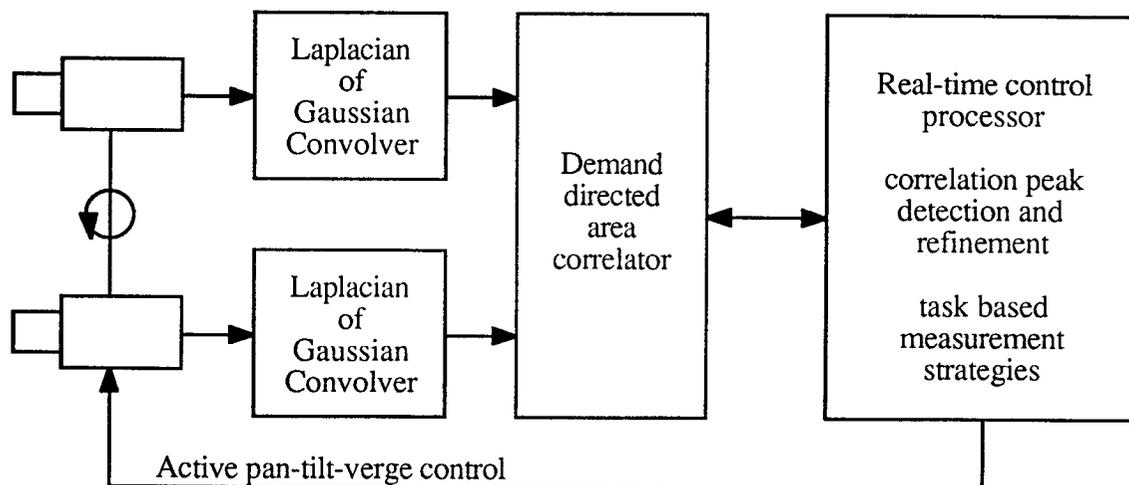

Figure 10: NFOV System Block Diagram

2. Search larger horizontal disparity ranges or search in vertical disparity as well by operating the correlator hardware through several cycles at the same image location.

3. Dynamically adjust the placement of range measurements to suit task needs such as looking for a range edge in a given locale or tracking a range feature as it moves.

The Prism-3 correlator hardware is also configured to allow correlations to be computed between successive frames from a single camera. This allows optical flow measurements to be made. In a tracking application, the system has been programmed to handle image velocities as large as 50 pixels per frame in any direction with subpixel measurement resolution.

6.3.2.3 The Active Camera Head

The pan-tilt mechanism used, shown in Figure 8, is a fairly straightforward device with orthogonal pan and tilt axes. Rather than having the tilt motor move with the pan axis, the tilt motor is stationary, with its drive shaft concentric with the pan axis. The tilt axis is then driven by a bevel gear set. The pan axis is simply driven by the motor through a set of spur gears. This arrangement with both motors stationary reduces the inertia of the head and simplifies the wiring. This arrangement also produces a coupling between the pan and tilt drive motors, such that if we wish to simply pan, both the pan and tilt motors must move simultaneously.

Each of the motor units is a DC brush motor with a 31:1 gear head, and a 500 line optical encoder mounted on the motor shaft. When using quadrature encoder counting, this results in an encoder sensing resolution of about 7 arc sec. for the pan and tilt axes.

In the initial mechanism tested, the motor gear heads had about 2 degrees of backlash, which is reduced to 0.5 degrees by subsequent gearing. This backlash was deemed unimportant because the camera weight tended to take out the backlash in almost all configurations.

Vergence Mechanism

The design of the vergence mechanism is very critical because of the strict repeatability requirements. It is important that the mechanism produce very high resolution with no uncompensated backlash.

Because of the limited range of motion, we chose to use what amounted to two symmetric four-bar linkages with flexible hinges forming the pivot points. The entire flexure is carved out of a single piece of nylon. The symmetry of the mechanism has a side effect of producing temperature compensation, so that the vergence angle will not change much with expansion or contraction of the nylon. The nut in the center of the mechanism is driven by a 20 threads per inch screw attached directly to the motor shaft. The vergence motor also has a 500 line encoder mounted directly on the motor shaft.

The baseline of the cameras is fixed at 22.2 cm., and the range of motion for each camera is from parallel to 10 degrees. This corresponds to a minimum target distance of about 60 cm.

The repeatability requirement for the vergence axis was determined for the case where the cameras were pointing at a distant object and were nearly parallel. With closer objects, the repeatability requirement is slackened considerably. Therefore, the vergence mechanism was designed such that with the cameras parallel, the mechanism is near a singular configuration. In other words, a very large motion of the drive screw is required for a very small change in the vergence angle, giving a very high control resolution. Even at the other end of the range of motion

with the cameras at maximum vergence, the encoders provide a resolution of better than 2.0 arc-sec.

Camera Mounting

The cameras need to be rigidly mounted on the vergence mechanism with stability on the order of the vergence control resolution. Any slippage or creep on these mounts larger than those values will necessitate recalibration of the camera alignment. It is also desirable to allow small ranges of adjustment in pitch and roll for each camera.

In our present design the cameras are attached to the out board edges of the nylon vergence mechanism by single 1/4 - 20 screws. The cameras are spaced away from the vergence plate by a short distance and two additional screws are used to push or pull on the camera body about an inch from the support screw for making small pitch and roll adjustments.

Motor Control

All three motors of the active head are controlled with closed loop servos, using the optical encoders for feedback. The control is implemented with dedicated motion control I.C.'s which read the encoders, accept command inputs from a host computer, and implement a PID control filter. These I.C.'s can be commanded in a raw position mode, a trapezoidal profile position mode with controlled velocity and acceleration, and in velocity mode. The pulse width modulated output of these I.C.'s go to an H-bridge amplifier to drive the motors.

Using stepper motors would have been a slightly less expensive alternative to using D.C. servos, but the D.C. servos allow the accelerations to be pushed to the motor limits without fear of losing steps as with a stepper drive.

For most non-tracking applications, we simply wish to move the head from one pan-tilt-vergence configuration to another, with some sort of smooth motion in-between. If we don't really care about the intervening trajectory, we can program each of the individual motor controllers to move to the final destination at a suitable maximum motor velocity and acceleration in their trapezoidal profile mode.

If we want all of the motors to start and stop at the same time, and if we want the head to move in a straight line in pan-tilt space, we must program the maximum velocity and acceleration of each controller such that each controller accelerates for the same period of time, slews at a constant velocity for the same period of time, and decelerates to the goal position for the same period of time. This can all be done with the trapezoidal profile mode, providing the motor control I.C. has enough resolution in its acceleration and velocity limits.

Velocity Control

For tracking applications, velocity control of the motors is more appropriate. The pan and tilt angular velocities as determined by the vision processor, undergo a simple linear transformation into motor speeds that are sent to the controllers. The motor command velocities can be continuously and asynchronously updated.

Tracking in depth is slightly more complicated in that the relationship between the tracking velocity in depth and the vergence velocity is dependent on the current depth of the target, and in turn, the relationship between the vergence velocity and the vergence motor velocity is dependent on the current vergence angle.

page 59

Given the tracking velocity in depth, the vergence motor velocity can be calculated either through a configuration dependent Jacobian matrix relating the two, or by simply differencing the inverse kinematic equations relating the target depth to the vergence motor angle.

Synchronization

When relying heavily on the trajectory profiling capabilities of the motor controllers, it is very important that we be able to synchronize the acquisition of images to the motion of the head. For example, we may wish to sweep the cameras at a constant velocity acquiring images at fixed angular increments, or conversely, to read the head position whenever an image is acquired.

The first implication of this is that the communication bandwidth to the motor controller for synchronization related commands should be as high as possible. For the VME bus controller board used in this system, this is not a significant problem. For controllers connected through an RS232 serial port, however, the control communication should be optimized so that only a few bytes at most need be sent to achieve synchronization.

The second implication is that the operating system must be sufficiently responsive to allow synchronization without critical processes being interrupted, or delayed in their execution. This is especially important if the active head control software is running as a process separate from the vision processing software.

Performance

Figure 8 shows the assembled pan/tilt/vergence head. The cameras are hung down below the vergence head in order to place the center of mass near the tilt axis. This reduces gravitational and inertial loading on the tilt motor.

The head can move through a 180 degree rotation in under a second and exhibits a positioning repeatability on the order of 50 arc seconds standard deviation in pan, 20 arc seconds in tilt, and 6 arc seconds in vergence.

Although performance of the vergence mechanism was quite repeatable, modeling the mechanism as a series of rigid links with isolated pivots did not provide commensurate accuracy. In fact, flexures used for hinges actually deflect along a short length, turning simple kinematic equations into bending beam problems. To compensate for this inaccuracy, we added an empirical vergence angle dependent correction factor to the nominal kinematic solution. We can make the spacing of these correction factors as fine as necessary to bring the accuracy as close to the repeatability as needed.

The peak velocities and accelerations, also met our goals. The real limitation in moving the head is in trying to decelerate the pan and tilt axes with only the camera weight to compensate the backlash. With higher accelerations, the camera head would bounce within the backlash as it was brought to a stop. The vergence axis, with no backlash, did not have this problem.

The other significant problem we encountered was with the resolution of the velocity and acceleration commands of the motor control I.C.'s used. It was difficult to get coordinated acceleration, slewing and deceleration on the pan-tilt axes using the simple trapezoidal position control modes provided by those chips. Newer versions of the control IC's should provide better control resolution and should solve this problem.

On the present system, we address this problem by doing more of the trajectory generation and control on the host processor. We use an interrupt service routine (ISR) that updates position goal points at a 60 Hz rate. This ISR software also supports velocity control of the head while ensuring that the head motions stay within the physical motion limits.

6.3.3 Color Vision System

The main components of the color vision system are the operations of building a model of a visual object and finding modelled objects in a novel image.

The basic approach to visual recognition used to support these operations involves representing images in the form of relational graphs, and matching these graphs to objects similarly represented. The relational graphs are automatically constructed over perceptually distinct image units, referred to as *blobs*. Given the relational graph representation, our strategy for object recognition is a voting-based search for partial matches of limited complexity.

6.3.3.1 General recognition strategy

Our basic recognition strategy has three aspects: (1) the use of relational graph representations of images and objects; (2) recognition by partial graph matching; and (3) the use of a voting process to determine the best partial match.

1. A relational graph of an image is composed of elements and arcs. The elements represent detectable units of image information, and the arcs between the elements represent geometric and other visually detectable relationships of the associated units. For this work, the units of image information are the blobs defined above, and the relations are functions of their color and geometry. Relational graphs can be used to represent information salient for object recognition in images undergoing a high degree of variation due to changes in viewing conditions and object articulation [14][15]. In this work, the images and the object models are both represented as relational graphs, and recognition is achieved by finding matches between these graphs.

2. A complete match between the model and image is generally impossible in real images, and the search for the most complete match is a complex problem that is not realistic for fast recognition. Fortunately, it is also generally not necessary for confident recognition; usually only a fraction of the relations representing the image have to agree with those representing the modeled object to unambiguously identify the object's projection in the image. Our strategy is to recognize the object in the image by searching for partial matches of limited complexity.

    Specifically, our strategy is to search for the best match of a single model element to a single image element. The best element-element match is defined to be the one with the most agreement between the modeled relations involving the model element and the image relations involving the image element, as represented in their respective graphs. This method effectively searches for the most complete matching subgraph that has a star topology: the single element considered the best match is the hub, and the supporting, matched relations with other elements are radiating arcs from this hub. In this work, the images and objects are modeled with completely connected relational graphs. Therefore,

there is generally substantial support for each correct single element match in the form of a partial, star-graph match involving all of the detected elements of the model. In addition, we argue that this support can be efficiently computed.

3. Our method of computing the support for a single element match is to accumulate votes for each single element match from each supporting (i.e., matching) relation, and then return the element-element match with the highest vote. Like other voting methods, such as geometric hashing [43] and generalized Hough transforms [9][13], the complexity of the process is polynomial with respect to the number of image elements. In our case, the relational graphs are composed of unary and binary relations. The recognition process visits every pair of image elements, and the complexity is $O(n^2)$ for n image elements. However, unlike most other recognition algorithms based on voting, the object to be recognized does not have to be rigid, or composed of elements that must be localized in the image with a high degree of reliability.

6.3.3.2 The relational graph representation

In the first step of recognition, blobs are extracted from the image and a relational graph is automatically constructed over them. Prior to recognition, this process is also followed to automatically construct a model of the object from one of its views.

The relational graph of an image is composed of unary and binary relations, involving one and two blobs, respectively. A relation has two attributes: a type of measurement (e.g., hue, distance or angle) and a value for that measurement. For every image processed, the relational graphs constructed are complete in the sense that all blobs are assigned a unary relation of every type and all pairs of blobs are assigned binary relations of every type.

The overall objective in designing the relational measurements is the same as for any feature selection problem encountered when developing a recognition system: the feature should tend to have a narrow distribution with respect to any one object and tend to differ in value from object to object. For relational measurements over the projections of three dimensional objects, this is achieved to a great extent by designing measurements that are relatively stable with viewing condition change and diverse in type. For this work, a color measurement invariant to surface orientation change and a relatively complete set of geometric measurements invariant to rotation, scaling and translation of the image (RST) are used.

A blob has color, position, length, width and orientation (when length and width are not equal). The aspect ratio (width over length) and hue of a blob are used as the unary relations. The aspect ratio is the only distinct geometric measurement of a single blob invariant to RST transformation in the image.

Assuming that the light source is roughly balanced in red, green and blue, the hue of the extracted blob is relatively unaffected by changes in the object's surface orientation relative to the light source and the amount of the light source the object is receiving. Ideally, the saturation component of the color is also unaffected; however, due to the strong specular component of the object surfaces involved in this project and the diffuse lighting of the scene, the perceived surface color saturation varies considerably.

The binary relations between two blobs express their relative orientation, size, and position (scaled to the size of the blobs). These measurements are also invariant to RST transformations [14]. The formulation of the actual measurements used here also takes into account the instability of the blobs extracted.

6.3.3.3 Computing votes from relational information

During the matching process, a value is assigned to a match between model relations and image relations. This value is mapped between 0 (bad) and 1 (good), and is used as a weight for the vote of the match.

For each relational measurement, the value is a function of the difference between the model and image measurement. The mapping to the (0, 1) range should reflect the differences possible due to blob detection error, when the match is actually correct: if the average difference for the correct match is large, then large difference should be assigned similar values as small difference. The following mapping is used from measurement difference to value: if the measurement difference d is less than the average when the match is correct a, then assign a value of 1; if $d < 2a$ assign a value of $2 - (d/a)$; and if $d > 2a$ assign a value of 0.

6.3.3.4 Blob extraction

The basic strategy applied in the blob extraction process is to isolate regions in the image that are homogeneous and distinct in some property, and then estimate the rough position, extent, orientation and average color of each region.

There are two basic factors involved in determining a perceptually distinct and homogeneous region in the image: the geometry of the region, and the statistics of the visual measurements within and without the region.

Geometrically, a region is most visually apparent when simple, compact and reasonably continuous in representation over some area in the image. In this work, for reasons of speed in implementation, the geometry of a region is considered suitable if it is continuous; more specifically, a region is some single, connected component of the image. The general approach to region formation developed here is that of growing and merging region fragments; by constraining regions to connected components, the process of selecting fragments for combination is greatly simplified. The approximation of a perceptually distinct region as a connected component produced useful representations of the image data for the recognition system discussed below.

Statistics of the visual measurements within and without the region are also important for determining the visual significance of a region. For this work, a region is considered visually significant if the difference between the average measurement in the region and the average measurement of each surrounding region is greater than some experimentally determined threshold.

The region formation algorithm was developed for quality and speed; it is a two-pass region grow and merge. The algorithm is a hybrid of one-pass region growing and iterative merging. In the first pass, the image is scanned, and for every sample it performed the following operations. If the sample is similar enough to a neighboring region, it is merged; if the sample turned out to be a

link connecting two regions that are similar enough, they are merged together; and if the sample is not similar to neighboring regions, a new region is formed initially containing only the sample.

This is followed by one more pass, where the regions of the previous pass are checked for neighbors that are similar enough; if so, they are merged.

The quality of the two-pass grow and merge algorithm is close to the slower and more involved iterative merge process. This indicates that, for a high percentage of cases, most of a region's extent can be extracted in two merging passes. In addition, its speed is comparable to the simpler one-pass region growing algorithm.

6.3.3.5 Color representation

In this work, the blobs are extracted with respect to image color measurements: a region is considered homogeneous if its parts are similar enough in color and distinct if the neighboring regions are different enough in color.

Hue is subjectively and experimentally found to be the most important factor; similarity in hue is used to determine if regions are to be merged or not. Hue is computed using the formula presented in [30]. Hue is unstable for pixels with very low saturation and brightness. For this reason color samples low in these values are ignored during region extraction.

6.3.3.6 Blob geometry

Once blobs are extracted from the image in the form of regions of samples, their basic geometric properties are measured. In this work, the principal model of recognition is of matching representations of geometric arrangements of parts. To achieve a basic representation of the geometric arrangement of blobs, it is useful to estimate their rough positions, orientations and extents. More specifically, the geometric attributes estimated for a blob are the position of the center of mass, the orientation of the principal axis, the length and the width.

The principal axis of the region point set can be successfully defined as the normal direction to the eigenvector of the region scatter matrix that is associated with its smallest eigenvalue. This method is discussed elsewhere [8][25].

The length and width can also be derived from the eigenvalues of the scatter matrix. The length is proportional to the square root of the largest eigenvalue, divided by the number of samples (minus one), and the width is similarly related to the smallest eigenvalue. The proportionality constant used depends on the nature of the sample distribution within the region. For a region with a roughly uniform sampling across its extent, the constant is estimated to be 4.

6.3.3.7 Color package recognition

Figure 11 shows the results of color package object recognition. the package is a pack of Big Red chewing gum. Using the COLOR-LEARN-OBJECT function from the testbed API, the color system learned the object as two red regions in an adjacent configuration. Once learned the COLOR-FIND-OBJECTS function retrieves a match for the learned pack of gum. This implementation worked well for the packages of candy used in the experiments described later in Section 7.

Figure 11: Color Package Recognition

6.3.4 Sound Player

The sound player was written in C using SunOS 4.2.1 sound toolkit capabilities provided by the Sun Sparc hardware. A TCX interface was defined for this sound player for which a TCX-CL/CLOS interface was provided for use by the testbed API and agent processes.

6.3.5 Customer Terminal

The customer terminal was implemented as emacs buffers operating within the testbed API Allegro CommonLISP/CLOS environment. Two buffers exist on the customer terminal: the Transaction Registry and the Customer Service Terminal. The interface to these two text entry buffers was described in Sections 3.7 and 5.4.

6.3.6 The Tracker

Tracking and control applications require fast low latency response from the sensor to be of value. A natural limit on speed is the frame rate of the camera system and for most cameras currently available this is standardized to between 1/30th and 1/60 of a second.

At 30 Hz, a person three meters from a camera walking across the field of view at 1 meter per second will traverse about 38 arc minutes per frame. With a 50mm lens the interframe motion disparity will be on the order of 30 pixels for a 512x512 image. This estimate is for one set of parameters — disparity magnitude varies approximately linearly with lens focal length, subject distance, subject speed, and frame rate — but it gives an indication of the kind of matching performance that will be required to follow human scale motions.

Similarly the head position control must be responsive to velocity commands at that 30Hz rate with maximum acceleration, and velocity limits sufficiently high to allow smooth pursuit tracking motions.

A system designed to meet these performance specifications was implemented in three subsystems, a low-level electronic tracking system, a mechanical servoing system, and a figure stabilization system. These individual mechanisms operate as parallel process threads which are loosely coupled. The electronic tracker makes high performance image based measurements of optical flow and stereo range and attempts to electronically follow an externally designated patch of surface so long as it remains within the camera field of view. The mechanical tracker operates the active camera head in velocity mode using a PID control algorithm. This system attempts to keep the head pointed so that the coordinates of the surface patch tracked by the electronic tracker are kept close to the center of the camera field of view. The figure stabilization submodule uses stereo measurements to assess the extent of the figure associated with the tracked patch. If the tracked patch is not centered on that figure, this module sends an error bias signal to the electronic tracker in an attempt to push it back to the center of the figure. This helps to maintain tracking on figures that are undergoing rotation that would otherwise lead an optical flow based tracking scheme astray.

6.3.6.1 The electronic tracker submodule

The electronic tracking module measures optical flow velocity and stereo range off of a single window position in the camera image. That window position is initially designated

externally. For example, the initial window coordinate might be set to the center of the camera field and started with the cameras pointed at a person to be followed. The Prism-3 correlator is configured to use six of its 36 correlators for stereo correlation and the remainder for measuring optical flow — that is correlations on successive images from the same camera.

A software control routine reapplies the correlator hardware 30 times with different disparity settings to measure correlation against motion at 900 different motion disparities (30 by 30 samples). The samples are spaced apart in disparity space by 4 pixels so this gives a total disparity range of 120 pixels. This means that any motion less than 60 pixels between camera frames will be covered within the search range.

Similarly, stereo correlations are done at 180 disparities (60 by 3 samples). With the four pixel sample spacing in disparity space, this covers a disparity range of 240 pixels horizontally by 12 pixels vertically. Thus as long as the actual disparity at the tracked patch is less than 120 pixels either way it will be detectable and vertical misalignments as large as 6 pixels can be tolerated.

The above set of 900 plus 180 correlation measurements is made each frame time. If a well-formed correlation peak — satisfactory height and sharpness — is located in the optical flow correlation surface that peak disparity is used to adjust the correlation window position to cause it to follow the measured motion for the next frame's measurements. The current tracking window coordinates are reported to the external process modules.

Likewise if a well formed stereo peak is identified, the disparity measurement — out of the current stereo fixation plane is reported.

6.3.6.2 The mechanical tracker submodule

The head servo mechanism is by its nature a more sluggish system that must cope with inertial mass, power and vibration constraints that limit its responsiveness. To minimize the influence of these limitations on tracking a moving image patch, the electronic tracker described in the previous section is only responsible for moving a massless window on the image. The mechanical servo system is made responsible for monitoring that window position and servoing as fast as it can to keep that window near the center of the camera field.

The mechanical servo operates in velocity control mode which is natural for this application. For example, if a tracked target is moving to the left across the camera field at 10 degrees per second, the head servo will see the tracked window move off to the left edge of the camera field and it will start to accelerate in that direction. As it does the tracked window will cease to move on the camera image since the head velocity is catching up with its velocity. A PID control algorithm is used to cause the head's turning velocity to exceed the velocity of the tracked entity sufficiently to get it back to the center of the camera field quickly with minimal overshoot.

The stereo disparity error reported by the electronic tracking module is used to drive the vergence control motor on the active head using a simpler proportional control algorithm to keep the cameras verged on the tracked target.

6.3.6.3 The figure stabilization submodule

The above tracking mechanism is responsive and tracks moving subjects fairly well. Errors in the optical flow tracking, however, are cumulative and the optical flow tracker is also fooled by objects rotating in place. To mitigate these types tracking effects without degrading the tracker's responsiveness a simple figure-ground discriminator was developed. The original implementation of this subsystem was provided by a collaborator, Eric Huber, working at the Johnson Space Center.

The figure-ground discriminator applies an additional set of 6 stereo disparity correlations covering a disparity range of 24 pixels at each of 16 locations on a 4 by 4 grid centered on the tracked window. This grid initially covers about half of the camera field. At each of the 16 locations checked, the stereo correlations are assessed to decide whether or not there is surface material within a disparity range of plus or minus 12 pixels. If so, it is assumed that the tracked figure extends out beyond that image position.

The center of mass of the figure as estimated on the 4 by 4 grid of samples is used to compute an offset bias that if large enough will be used to readjust the tracked window position.

In addition the approximate size of the tracked figure is computed from its image on the 4 by 4 sample grid. If the grid spacing is too large compared with the estimated object size, the dispersion of the grid is reduced and vice versa.

6.3.6.4 Performance and enhancements

The current implementation of the tracker module achieves fairly good tracking performance. In an office environment, it is able to follow an individual walking at a brisk pace at ranges from 20 feet in to about 4 feet and over the active heads full pan and tilt ranges.

The tracking system is designed to be started on an object to be followed and it has no ability to maintain object identity across occlusions — such as a target walking behind another person. These kinds of enhancements are the subject of current research.

We are also investigating mechanisms for initially directing the attention of the tracker onto objects. Our initial ideas about this include scanning the visual space for things that have disparate motion or that have changed in range from a prior baseline.

6.3.7 Customer Detection Behavior

An integrated action (behavior) was developed to detect customers at the service counter. This behavior sequences the active head movement, PRISM-3 depth measurements, and gantry position to scan the counter, find a customer torso, and scan to center the customer head in the field of view. The customer detection API was described in Section 5.5.5.

The behavior moves the gantry torso to a fixed position and directs the active head to scan the visual area just above and beyond the service counter. While the head is in motion, stereo measurements are made continuously in a 10x10 grid evenly placed over the 512x512 stereo camera image. When a customer is standing at the counter, the stereo measurements reflect their presence in the field of view. Because people may stand within the field of view, but not *at* the service counter, a person is considered to be standing at the service counter only when they are

within about 1.5' of the service counter. Depth measurements within 1.5' of the service counter are deemed *in-range*.

Once a customer torso has been detected, the behavior attempts to find the customer head. Because many stereo measurements can be very quickly performed, a simple and effective strategy can be employed. The pan and tilt of the active head are independently servoed. The pan position is servoed to center the centroid of in-range horizontal depths relative to the horizontal center of the image. The tilt axis control moves the visual gaze to the top of the customer head. The top of a customer head is defined using the 10x10 depth measurement array when the top row depths are out of range while the other rows contain in-depth measurements. When the top row contains in-range measurements, the head is looking at the customer torso, and the head must tilt up. A fast and natural way to do this is to tilt the head up at a fixed velocity until the top of the head moves into the field of view, and this velocity must be slow enough to allow a depth measurement to be made while the head is in view. Alternatively, head tilt up may be determined in other ways (e.g., by one quarter of the field of view, by customer height probability distributions). When more than one row at the top of the depth measurement array is not in-range, the head is looking above the customer head, and the head must tilt down. The required head tilt down is computed from the number of out-of-range depth measurement top rows and the vertical angle subtended by a single depth measurement row. Head tilt position remains unchanged when only the top row is not in-range and the other rows are in-range. Customer head detection may be considered achieved when the customer head is horizontally centered and the top of the head is at the top row of the depth measurement array.

The customer detection behavior executes within Allegro CommonLISP/CLOS, and head commands and PRISM-3 depth measurements are made via the testbed API which communicates across Ethernet. With this configuration, a closed loop control of 6-10Hz was achieved. Unless a customer continuously moves a large amount while standing at the service counter, this servo rate worked well in reliably finding a customer and moving the visual field to their head. At this point, the agent may choose other actions (e.g., turn the tracker on to monitor the customer head position at a 60Hz rate, move the torso up to the customer to initiate a package transaction).

7 EXAMPLE AGENT EVALUTION IN THE PC-Serbot TESTBED

7.1 Experimental Agent

Using the Testbed API implementation described in Section 6, a simple example agent was programmed. This agent was designed to operate in the package handling domain as described in Section 3.4 in which the packages are candy (e.g., packages of bubble gum, a PEZ dispenser). The testbed implementation provided for stereo vision, color vision for package recognition, movement in the workspace, and arm/end-effector package handling capabilities. Using these perception and motor skills, a simple programmed agent was programmed in CommonLISP to be able to perform extended package counter benchmark task description (BT2) requirements that were described in Section 3.6.2. This simple agent met BT2 requirements with the exceptions that: the storage areas are initial empty, customers perform only one transaction at a time, and there are limited abilities to overcome errors in sensorimotor systems. This simple agent was used to demonstrate operation in the PC-Serbot domain.

The sample agent successfully carried out the benchmark tasks including the more difficult task BT2. During execution, operating characteristics were measured by the time spent in particular agent activities, and these timings were used to compute agent performance measures related to the metrics described in Section 4. Apart from demonstrating the successful operation of the physical Testbed API implementation, the metrics computed from this agent suggested how more efficient agents could be designed.

The example candy package agent scenario was as follows:

- The agent uses the stereo system to continuously scan the counter area for customers. When a customer is found, it directs the head to look at the customer's face.

- The agent then approaches the customer and greets them (i.e., plays "Welcome to Teleos").

- The agent then enters into a customer dialog to determine the customer's desired transaction. It does this by:

  - *Extending its gripper and playing "Please place your package or prototype in my hand." Using the force-torque sensor to determine when the customer has placed the candy in its gripper, the agent then takes the package, puts it down, and directs the color camera at the package. If the package has not been seen before (i.e., the color vision system has no match to an object in the storage area), the agent concludes that the customer is dropping off the package. Otherwise, if the color system provides a match to an object already placed in the storage area, the agent then concludes the customer wants to pick-up that type of candy from the storage area (much like getting movies from a video rental store).*

- The agent then executes the transaction:

  - *For a drop-off, the agent plays "Thanks for the package, have a nice day," and it then moves to the storage area. The agent moves systematically through the storage area looking at placement surfaces, and it runs the color object recognition system to determine if the surface is clear to place down a package (i.e., when a storage area is seen for which the color vision system does not match any objects). Upon finding an*

open position, the agent puts the package down, steps back to look at the package, it invokes the color object learning system, and it indexes the placed position of the object with its object model for future reacquisition.

- *For a pick-up, the agent returns the package the customer gave as an example, and it plays "Oh, you're here to pick up a package. Please wait here while I get your package." Using the object model index matched by the color vision system, the agent moves to the last known position of the package to pick it up. No visual reacquisition of package position or closed-loop hand-eye grasping is performed. If a package is picked-up (i.e., something is grasped), it is returned to the customer. The agent plays "Thanks for the package, have a nice day, Hope I got the right package!"*

- The agent resumes monitoring the counter area.

7.2 Analysis of Agent Performance

The agent described above performed 11 drop-off and 10 pick-up transactions in which there were no packages in the storage area in the initial configuration. During the execution of the agent, activity and timing information was logged to a file. This logged information included:

- **Object sensing.** The begin and end times for color vision system calls to perform object sensing and matching.

- **Customer dialog.** The begin and end times for customer dialog operations such as awaiting a package handover from the customer and playing sounds.

- **Movement.** The begin and end times for all physical agent movements such as moving the gantry, arm, gripper or camera head.

- **Transaction.** The time from when a transaction begins (i.e., a customer is detected) to when a transaction has been completed.

The log file was used to compute where agent efforts were spent during the execution of drop-off and pick-up transactions.

Figure 12 shows drop-off transaction execution times for the 11 drop-off transactions performed by the agent, and Figure 13 shows execution times for the 10 pick-up transactions. Notice that the transaction time was dominated by the physical movement of the agent. This time was spent in moving up to the customer, picking up and putting down packages, and examining the storage area. Customer dialog largely was dominated by the time required to interact with the customer to receive a drop-off package or pick-up prototype. As can be seen, object sensing did not require substantial time: simply to look at a customer package to determine the transaction type, and to examine open storage area surfaces. Each such color vision system call took about 2 seconds to execute (on a SparcStation LX).

Figure 14 shows comparative operation execution times for drop-off and pick-up transactions. Note that pick-up transactions take more time — about 38 seconds on average. Most of this additional time is spent in the additional movement a pick-up entails: the package prototype is handed over, looked at, and returned back to the customer before proceeding to pick-up the object from the storage area. In addition, a small additional amount of customer dialog is required for pick-ups to effect package transfers.

A more opportunistic or efficient agent could have achieved more effective use of resources. The agent execution times shown in these figures were measured in a benign environment in which packages and grappling had good certainty, gantry and arm positional errors were not large, packages stayed where they were put, and nothing was dropped. An agent that was more robust against such unexpected events would increase the time spent object sensing in order to overcome these potential problems.

The simple example agent did not consider how activities could be interleaved to achieve greater utilization of computational resources to improve performance. This can be seen in Figure 12 and Figure 13 since the overall transaction execution times are roughly the sum of the time
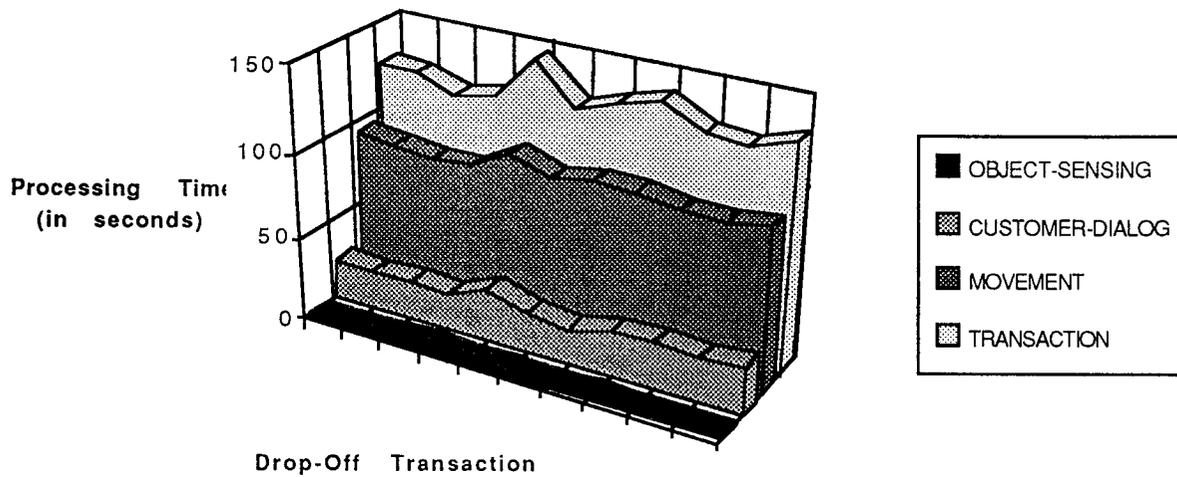
Figure 12: Drop-Off Transaction Execution Times (11 transactions)
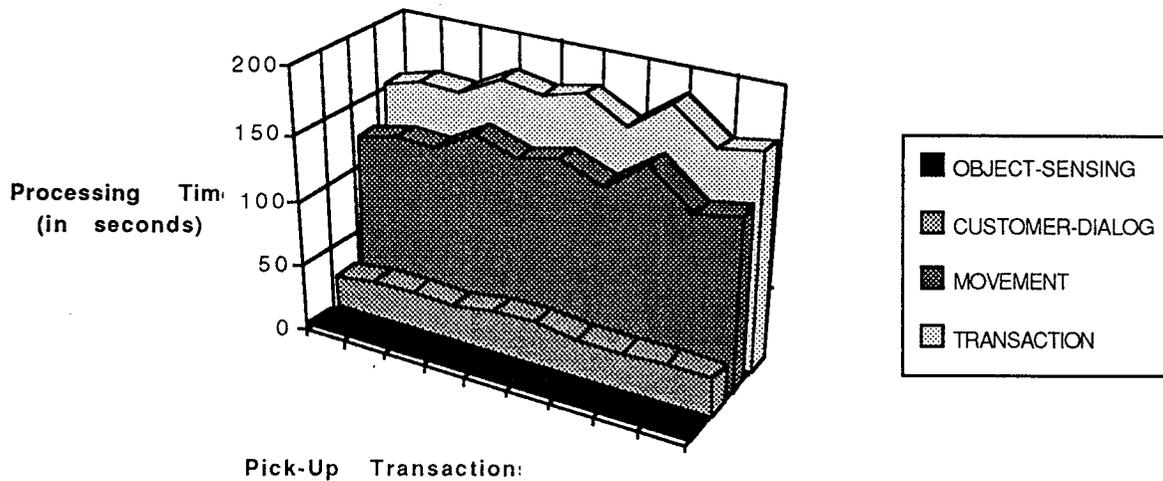
Figure 13: Pick-Up Transaction Execution Times (10 transactions)

spent in subactivities. Since concurrent computing resources are available to the agent — the PRISM3, color vision system computer, gantry controller, and agent computer — agents can achieve far greater utilization of available resources. In addition, an agent that processes multiple transactions at one time can gain opportunistic efficiencies by avoiding duplicated efforts (e.g., moving to a storage area to pick up two packages at once).

Figure 14: Comparative Drop-Off and Pick-Up Operation Execution Times

8 SUMMARY

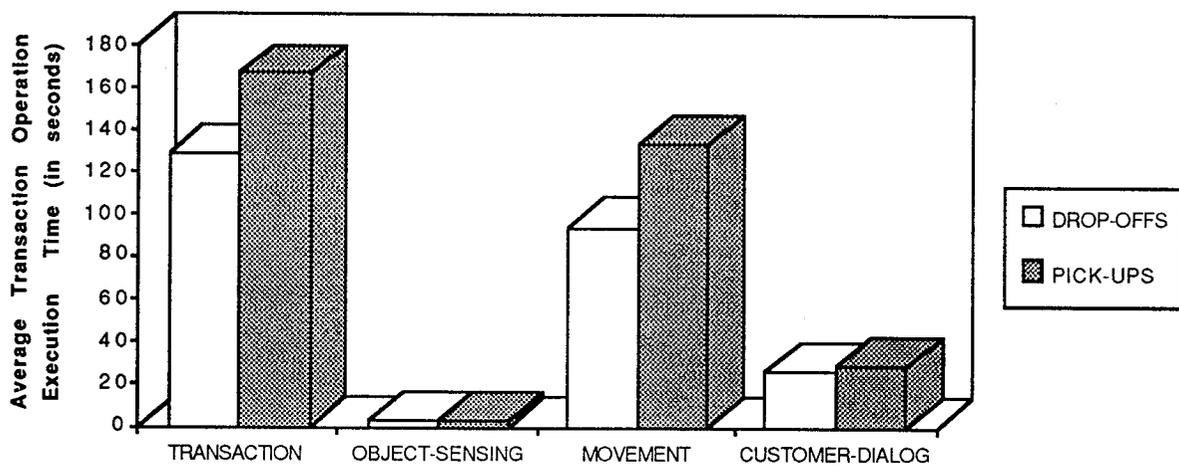Evaluation of integrated agent architectures operating in realistic domains is important for understanding and guiding research and application of these agents. The work presented in this report provides a better understanding of how such evaluations may be made and how future work may be guided.

An important contribution of this work was the development of a canonical domain definition for which agents and methodologies can be tested and evaluated. The relationship of this domain to well-understood planning problems lends it continuity with classical analyses. In addition, it provides areas of complexity for which differences between planning techniques and reactive agent methodologies become apparent.

In addition to the canonical problem domain, a set of evaluation metrics and criteria were proposed that form a common basis for comparison between agent construction methodologies. By using these techniques, agent performance may be quantitatively measured and compared. In the past, such comparisons have been difficult to make, especially for agents operating in physical domains.

This work provided a specific Testbed Application Programmer's Interface (API) definition that an agent designer can use to build an effective agent. This API describes the base repertoire of skills available to an agent and these skills must support operation in a real and relatively complex domain. This Testbed API provides an interesting partitioning of agent skills that can be applied in other operational domains.

The implementation of the Testbed API required the development of a broad set of real-time skills including stereo and color vision, robotic arm, camera and body movement, force/torque feedback, and detection and interaction with customers. These skills were integrated into a seamless software implementation that executed over several heterogeneous, distributed computers. The implementation of the Testbed API is noteworthy in the degree of advanced capabilities brought to bear for an agent, as well as the manner in which these capabilities were integrated and made to interoperate.

In addition to the accomplishments and advances made within this project, the IAA Project has stimulated agent evaluation activity within the community. This has been demonstrated in part by Teleos' sponsoring of the Benchmarks and Metrics Workshop (see Appendix A.) and leadership in establishing the National Virtual Laboratory (see Section 2.3.2).

Further work is needed to better develop the Testbed API to expand the range of applicable problem domains and derive additional agent evaluation metrics. Activities related to the National Virtual Laboratory effort afford natural opportunities to expand and apply the capabilities developed under this project.

9 REFERENCES

[1]   P.E. Agre and D. Chapman, "Pengi: An Implementation of a Theory of Activity," AAAI-87.

[2]   P.E. Agre, "Routines", MIT AI Memo 828, May 1985.

[3]   J. Aloimonos, I. Weiss, and A. Bandyopadhyay, Active vision, In *International Journal of Computer Vision*, number 1, pages 333—356, 1988.

[4]   R. Alterman, "Adaptive Planning", Cognitive Science, vol. 12, pp. 393-421, 1988.

[5]   R.C. Arkin, "Integrating Behavioral, Perceptual, and World Knowledge in Reactive Navigation," Robotics & Autonomous Systems, vol. 6, nos. 1 & 2, pp. 105–122, June 1990.

[6]   R. Bajcsy, Active perception. In *Proceedings of the IEEE*, volume 76, pages 996—1005, 1988.

[7]   D. Ballard, Animate vision, *Artificial Intelligence*, 48:57—86, 1991.

[8]   D. Ballard, *Computer Vision,* Prentice Hall, Englewood Cliffs, NJ, 1982.

[9]   D. Ballard, "Generalizing the Hough Transform to Detect Arbitrary Shapes," in *Pattern Recognition,* vol. 13, no. 2, pp. 111-122, 1981.

[10]   M.E. Bratman, D.J. Israel, and M.E. Pollack, "Plans and Resource-Bounded Practical Reasoning," *Computational Intelligence*, vol. 4, no. 4, 1988.

[11]   R.A. Brooks, "A Robust Layered Control System for a Mobile Robot," IEEE J. Robotics & Automation, vol. RA-2, no. 1, pp. 14–23, March 1986.

[12]   C. Brown and R. Nelson, "Image understanding research at Rochester", *IUW* 1990, pp. 76.

[13]   T. Bruel, "Fast Recognition Using Adaptive Subdivisions of Transformation Space," *Proc. CVPR,* pp. 445-451, 1991.

[14]   J.B. Burns, R. Weiss, and E.R. Riseman, "View Variation of Point-set and Line-segment features," to appear in *IEEE Transactions on Pattern Analysis and Machine Intelligence,* January 1993.

[15]   J.B. Burns, and E.R. Riseman, "Matching Complex Images to Multiple 3D Objects using View Description Networks," *Proceedings: IEEE Conf. on Computer Vision and Pattern Recognition,* June 1992.

[16]   P. Burt, Smart sensing with a pyramid vision machine. In *Proceedings of the IEEE,* number 76, pages 1006—1015, 1988.

[17]   J.H. Connell, "A Behavior-Based Arm Controller", IEEE Transactions on Robotics and Automation, vol. 5, no. 6, pp. 784-791, December 1989.

[18]   P.R. Cohen, A.E. Howe and D.M. Hart, "Intelligent Real-Time Problem Solving: Issues and Examples," *AI Magazine,* vol. 10, no. 3, pp. 32-48.

[19]   P.R. Cohen & H.J. Levesque, "Intention is Choice with Commitment", *Artificial Intelligence,* vol. 42, pp. 213-261, 1990.

[20]   D. Coombs and T.J. Olson, Real-time vergence control for binocular robots. *IJCV,* 1991.

[21]   D. Coombs and C. Brown, "Real-time smooth pursuit tracking for a moving binocular robot", *CVPR* 1992, p. 23.

[22]   Y. Descotte and J. Latombe, "Making Compromises Among Antagonistic Constraints in a Planner," Artificial Intelligence, vol. 27, 1985.

[23]   M. Drummond, "Situated Control Rules," In R.J. Brachman, H.J. Levesque, R. Reiter (eds), *Proc. 1st Intern. Conf. Principles of Knowledge Representation & Reasoning,* Morgan-Kaufman, Los Altos, CA, 1989.

[24]   M.E. Drummond and L.P. Kaebling, "Integrated Agent Architectures: Benchmark Tasks and Evaluation Metrics," TR-90-08, Teleos Research, Palo Alto, CA, 1990.

[25]   R. Duda and P. Hart, *Pattern Classification and Scene Analysis,* Wiley-Interscience, 1973.

[26]   C.L.Fennema, Jr., "Interweaving Reason, Action and Perception", COINS TR91-56, University of Massachusetts at Amherst, September 1991.

[27]   N.J. Ferrier, The Harvard Binocular Head, Technical Report 91-8, Harvard Robotics Laboratory, 1991.

[28]   R.E. Fikes, P.E. Hart and N.J. Nilsson, "Learning and Executing Generalized Robot Plans," *Artificial Intelligence,* vol. 3, no. 4, pp. 251-288, 1972.

[29]   R.J. Firby, "Adaptive Execution in Complex Dynamic Worlds", YALEU/CSD/RR #672, January 1989.

[30]   J. Foley, A. van Dam, S. Feiner and J. Hughes, *Computer Graphics: Principles and Practice,* Addison-Wesley, 1990.

[31]   M. P. Georgeff and A. Lansky, "Reactive Reasoning and Planning," In *Proceedings AAAI-87,* pp. 677–682, AAAI, 1987.

[32]   P. Haddawy & S. Hanks, "Issues in Decision-Theoretic Planning: Symbolic Goals and Numeric Utilities", Proceedings: Workshop on Innovative Approaches to Planning, Scheduling and Control, pp. 48-58, November 1990.

[33]   B. Hayes-Roth, "Architectural Foundations for Intelligent Agents," *The International Journal of Real-Time Systems,* vol. 2, pp. 99-125, 1990.

[34]   C. Hewitt, "The Challenge of Open Systems," Byte, vol. 10, no. 4, pp. 223—242, April 1985.

[35]   W. Jacobs & M. Kiefer, "Robot Decision Based on Maximizing Utility", Session No. 15 Robot Problem Solving, pp. 402-411.

[36]   L.P. Kaelbling & N.J. Wilson, "Rex Programmer's Manual," Technical Note 381R, SRI International, Menlo Park, CA, July 1988.

[37]   L.P. Kaelbling & S.J. Rosenschein, "Action and Planning in Embedded Agents," Technical Report, Teleos Research, Palo Alto, CA, November 1989.

[38]   P. Kahn, "Building Blocks for Computer Vision Systems", *IEEE Expert,* to appear.

[39]   P. Kahn, "Specification & Control of Behavioral Robot Programs," *SPIE Sensor Fusion IV: Control Paradigms & Data Structures,* Boston, MA, Nov. 1991.

[40]   E. Krotkov, *Active Computer Vision by Cooperative Focus and Stereo.* Springer Verlag, New York, 1989.

[41]   J.E. Laird and P.S. Rosenbloom, "Integrating Execution, Planning, and Learning in Soar for External Environments," *Proc. 8th National Conf. on Artif. Intell.,* pp. 1022-1029, 1990.

[42]   T.J. Laffey, P.A. Cox, J.L. Schmidt, S.M. Kao & J.Y. Read, "Real-Time Knowledge-Based Systems", *AI Magazine,* pp. 27-45, Spring 1988.

[43]   Y. Lamdan and H. Wolfson, "Geometric Hashing: A General and Efficient Model-Based Recognition Scheme," *Proceedings of the ICCV,* pp. 238-249, 19888.

[44]   D.M. Lyons, R. Vijaykumar, S.T. Venkataraman, "A Representation for Error Detection and Recovery in Robot Task Plans", Philips Labs-Briarcliff, Doc. no. TN-89-163, MS-89-113, December 8, 1989.

[45]   D. Marr and T. Poggio. A computational theory of human stereo vision, *Proceedings of the Royal Society of London: B,* 204:301—328, 1979.

[46] L.H. Matthies, Stereo vision for planetary rovers. Technical Report JPL D-8131, Jet Propulsion Laboratory, California Institute of Technology, January 1991.

[47] J.H. Munson, "Robot Planning, Execution, and Monitoring in an Uncertain Environment", Session No. 8 Robots and Integrated Systems, pp. 338-349.

[48] A. Newell and H.A. Simon, "GPS: A Program that Simulates Human Thought," in E.A. Feigenbaum and J. Feldman (eds.), *Computers and Thought*, pp. 279-293, R. Oldenbourg KG., 1963.

[49] N.J. Nilsson, "Action Networks," In J. Tenenberg, J. Weber & J. Allen (eds), *Proceedings from the Rochester Planning Workshop: From Formal Systems to Practical Systems*, pp. 36–68, 1989.

[50] H. Keith Nishihara, "Minimal Meaningful Measurement Tools", Teleos Technical Report TR-91-01, Teleos Research, Palo Alto, CA, 1991.

[51] H.K.Nishihara, "Tests of a sign correlation model for binocular stereo," *Investigative Ophthalmology and Visual Science*, vol. 30, 3, 389, March 1989.

[52] H.K.Nishihara, "Hidden information in transparent stereograms," *Proc. of the Twenty-First Asilomar Conf. on Signals, Systems, and Computers*, Pacific Grove, CA, 695-700, Nov. 2-4, 1987.

[53] H.K.Nishihara, "Practical Real-Time Imaging Stereo Matcher," *Opt. Eng.*, vol. 23, 5, 536-545, Sept.-Oct. 1984. Also in *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms*, edited by M.A. Fischler and O.Firschein, Morgan Kaufmann, Los Altos, 1987.

[54] C.A. O'Reilly & A.S. Cromarty, "'Fast' is not 'real-time': Designing effective real-time AI systems", SPIE, vol. 548, Applications of Artificial Intelligence II, pp. 249-257, 1985.

[55] D.W. Payton, J.K. Rosenblatt, & Dm.M. Kiersey, "Plan Guided Reaction," IEEE Trans. Systems, Man, & Cybernetics, vol. 20, no. 6, Nov./Dec. 1990.

[56] P.R.H. Place, W.G. Wood, M. Tudball, "Survey of Formal Specification Techniques for Reactive Systems", CMU/SEI-90-TR-5, ESD-TR-90-206, May 1990.

[57] M. Pollack and M. Ringuette, "Introducing the Tileworld: Experimentally Evaluating Agent Architectures," *Proceedings of the Eighth National Conference on Artificial Intelligence,* pp. 183-189, 1990.

[58] S.J. Rosenschein and L.P. Kaelbling, "Action and Planning in Embedded Agents," In *Robotics and Autonomous Systems*, vol. 6, 1990.

[59] S.J. Rosenschein and L.P. Kaelbling, "Integrative Planning and Active Control," *Proceedings of the NASA Conference on Space Telerobotics*, Pasadena, California, 1989.

[60] S.J. Rosenschein, "Formal Theories of Knowledge in AI and Robotics," in *New Generation Computing*, vol. 3, no. 4 (special issue on Knowledge Representation), Omasha, Ltd. Tokyo, Japan, 1985.

[61] S.J. Rosenschein, "Plan Syntheses: A Logical Perspective," *Proceedings of Seventh International Joint Conference on Artificial Intelligence*, Vancouver, B.C., August, 1981.

[62] M.J. Schoppers, "Representation and Automatic Synthesis of Reaction Plans," Ph.D diss., Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign, 1989.

[63] M. Soldo, "Preplanned and Reactive Control for Mobile Robot Navigation," IEEE Conference on Robotics & Automation, 1990.

[64] M. Swain and M. Stricker, "Promising Directions in Active Vision", *NSF Active Vision Workshop* (written by attendees of workshop), U. of Chicago, 1991.

[65] M.J. Swain, "Color Indexing," *TR 360,* University of Rochester, Computer Science Dept., Rochester, NY, Nov. 1990.

[66]   J. Woodfill and R. Zabih, Using motion vision for a simple robotic task. In *Sensory Aspects of Robotic Intelligence, Working Notes, AAAI Fall Symposium Series*, Asilomar, California, November 1991.

[67]   *Behavioral Architecture for Robot Tasks (BART): BART Language Software System User's Manual,* Advanced Decision Systems, a division of Booz•Allen & Hamilton, Inc., Mountain View, CA, July 1992.

[68]   *Teamworks Final Scientific Report,* Advanced Decision Systems, a division of Booz•Allen & Hamilton, Inc., Mountain View, CA, July 1992.

[69]   R.L. Marks, T.W. McLain, D.W. Miles, S.M. Rock, G.A. Sapilewski, and H.H. Wang, *Monterey Bay Aquarium Research Institute/Stanford Aerospace Robotics Laboratory Joint Research Program: Summer Report 1992,* Monterey Bay Aquarium Research Institute, Pacific Grove, CA 93950.

Appendix A. The First Benchmarks and Metrics Workshop

In support of the IAA Program, Teleos Research in conjunction with NASA and DARPA organized *The Benchmarks and Metrics Workshop* which was held at NASA Ames, California on June 25, 1990. This Appendix contains the *Collected Notes from The Benchmarks and Metrics Workshop* and a resulting paper by M.E. Drummond and L.P. Kaelbling entitled "Integrated Agent Architectures: Benchmark Tasks and Evaluation Metrics," *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 408-411, Morgan Kaufman Publishing, San Mateo, CA, November 1990 (see also TR-90-08, Teleos Research, Palo Alto, CA, 1990).

In addition, the work described in the following Teleos Research technical reports received support under the IAA project:

TR-90-06
Intermediate Vision: Architecture, Implementation and Use
David Chapman
October, 1990

TR-90-07
Specifying Complex Behavior For Computer Agents
Leslie Pack Kaelbling
December 1990
(See also *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 433-438, Morgan Kaufman Publishing, San Mateo, CA, November 1990.)

TR-90-10
Compiling Operator Descriptions Into Reactive Strategies Using Goal Regression
Leslie Pack Kaelbling
December, 1990

TR-90-11
Learning From Delayed Reinforcement In a Complex Domain
David Chapman and Leslie Pack Kaelbling
December, 1990

TR-91-01
Minimal Meaningful Measurement Tools
H. Keith Nishihara
October, 1991

TR-93-06
A Situated View of Representation and Control
Stanley J. Rosenschein and Leslie Pack Kaelbling
December, 1993

Integrated Agent Architectures:
Benchmark Tasks and Evaluation Metrics *

Mark E. Drummond
Sterling Federal Systems
NASA Ames Research Center
MS: 244-17, Moffett Field, CA 94035

Leslie Pack Kaelbling
Teleos Research
576 Middlefield Road
Palo Alto, CA 94301

1 Introduction

An *integrated agent architecture* is a theory or
paradigm by which one may design and program in-
telligent agents. An intelligent agent is a collection of
sensors, computers, and effectors, structured in such
a way that the sensors can measure conditions in the
world, the computers can process the sensor informa-
tion, and the effectors can take action in the world.
Changes in the world realized by the effectors close
the loop to the agent's sensors, necessitating further
sensing, computation, and action by the agent.

In recent years there has been a proliferation of pro-
posals in the AI literature for integrated-agent archi-
tectures. Each architecture offers an approach to the
general problem of constructing an integrated agent.
Unfortunately, the ways in which one architecture
might be considered better than another are not al-
ways clear.

For instance, Nilsson's (1988) *action nets* provide
a means for structuring actions in terms of the indi-
vidual goals they are to achieve in the environment.
Rosenschein and Kaelbling's (1986, 1989) *situated au-
tomata theory* provides a new view on the role of logic,
complexity, and information in situated agents, and
has resulted in a new generation of software tools for
building complex systems (Kaelbling, 1987a,b, 1988).
Schoppers (1987) has suggested an approach to plan
generation and execution based on the idea of *uni-
versal plans*. Georgeff and Lansky's (1985) *Procedural
Reasoning System* provides a graphical programming
environment based on the theory of augmented tran-
sition networks. *Plan nets* (Drummond, 1989; Drum-
mond & Bresina, 1990) act as generators of possible
behaviors and help explain the relationship between
planned and unplanned action. Recent work on the
SOAR architecture has studied the problems that arise
when an integrated learning and problem-solving sys-
tem interacts with an external environment (Laird &
Rosenbloom, 1990).

But so what? What can systems based on these
architectures really do? What can one do that another
cannot?

There has been a growing realization that many of
the positive and negative aspects of an architecture
become apparent only when experimental evaluation
is performed and that to progress as a discipline, we
must develop rigorous experimental methods. In addi-
tion to the intrinsic intellectual interest of experimen-
tation, rigorous performance evaluation of systems is
also a crucial practical concern to our research spon-
sors. DARPA, NASA, and AFOSR (among others) are
all actively searching for better ways of experimentally
evaluating alternative approaches to building intelli-
gent agents.

One tool for experimental evaluation involves test-
ing systems on benchmark tasks in order to assess
their relative performance. As part of a joint DARPA-
and NASA-funded project, NASA-Ames and Teleos
Research are carrying out a research effort to estab-
lish a set of benchmark tasks and evaluation metrics
by which the performance of agent architectures may
be determined. This paper is a short report on the
project's general aims and proposed methods. Possible
points of debate are addressed by looking back over the
transcripts of the Benchmarks and Metrics Workshop,
held at NASA-Ames in June, 1990 (referred to here-
after as the BMW-I). This paper does not reproduce
any statements from the BMW-I in verbatim form,
but instead attempts to communicate the essence of
the participants' comments.

2 Roles of Benchmark Tasks in the Research Community

Consider the following representative definition of
"benchmark."

> **Benchmark** *n* – surveyor's reference mark
> for determining further heights and dis-
> tances. (Garmonsway, 1965)

A benchmark is a reference point; a tool for deter-
mining where one stands. In this sense, a benchmark
does not uniquely determine how data obtained from
the study of that benchmark will be used. Data ob-
tained from the study of benchmarks can be used in
a variety of ways, and some of these are presented in
this section. In summary, we feel that in order to cor-
rectly determine "further heights and distances" in the

integrated agent research community, we must have a common language for describing agent architecture performance. One way to do this is by establishing a common set of benchmark tasks and evaluation metrics.

A suite of carefully-designed benchmark tasks would serve two primary roles in the research community.

From experiments to principles. The first, and most important, role would be to provide a common frame of reference for researchers. In the literature, a wide variety of incommensurable vocabularies are used to describe intelligent systems. Someone faced with the task of understanding the relative strengths and weaknesses of different architectures must, currently, be able to grasp the relationship between "productions," "wires," "operator descriptions," "problem spaces," and many other such concepts. A set of benchmark tasks would allow the performance of systems to be directly compared; in addition, it would allow the designers of agent architectures to relate the internal characteristics of their architectures to externally observable properties of instances of those architectures. A researcher would be able to say things like "System X performs as it does on benchmark A because its representation of time is so flexible but its algorithm for Y is too slow."

The understanding gained from such experiments is critical. Consistent success or consistent failure on a specific set of benchmark problems should lead one to consider what features the given set of benchmarks share. If some particular problem feature can be implicated in the necessary success or failure of a given architecture then this source of knowledge can be fed back into the architecture's design. Without this sort of "closed-loop" experimental evaluation, corrections to any given architecture can be motivated only by abstract mathematical and computational aesthetics. The space of possible architectures is huge, and most notions of formal aesthetics are ill-defined. It makes more sense to explore the space of possible architectures driven by success and failure on particular representative benchmark problems.

Enabling technology transfer. The occasional practical application of integrated agent technology would not hurt the field. A wide variety of practical industry and government problems would benefit from better technology transfer. Technology transfer would be facilitated by benchmark problems that are representative of the intrinsic difficulty of practical applications. As it stands, a person with a practical application task in mind is given no easy access into the relevant set of technologies. For instance, given the problem of controlling a particular device in a factory, under particular constraints and resources, which architecture would be more appropriate: PRS (Georgeff & Lansky, 1985) or O-Plan (Currie & Tate, 1985)? Which system is better suited for the application at hand? Both PRS and O-Plan have been tested on representative tasks: PRS has been applied to the Space Shuttle reaction control system and O-Plan has

been applied to spacecraft mission sequencing. However, not all architectures have been applied in this way, and even PRS and O-Plan must be further applied to other problems to better understand their individual strengths and weaknesses. The existence of a common set of benchmark tasks will make such comparisons possible across the entire integrated-agent research community.

As well as facilitating technology transfer to problems of practical interest, a common set of benchmarks can make new research ideas available to researchers in other fields. For instance, one might expect that research in the area of real-time operating systems would benefit from an understanding of the latest ideas in integrated agent architectures. Representative tasks of common merit would facilitate communication among various fields.

3 What Benchmarks Are Not

There are some obvious and some non-obvious worries associated with the establishment of a common set of benchmark tasks and evaluation metrics. This section briefly addresses some worries that were articulated by participants of the BMW-I.

Benchmarks will necessarily force us to worry only about numbers. A benchmark simply provides system performance indicators, perhaps expressed as numbers, perhaps not. These performance indicators can be used in a variety of ways. A particularly pedestrian use of performance indicators would be simple comparison; for instance, a question such as "which system got the highest score on task A?" with no further scrutiny as to *why* is deeply uninteresting. As mentioned above, the availability of performance indicators derived from the application of a particular architecture to a particular benchmark is a starting point for understanding the principles underlying the architecture's performance.

The benchmarks will not include all theoretically interesting problems and will also not be representative of practical, real-world problems. What if the benchmarks are so badly chosen that the performance data they engender is totally meaningless, both theoretically and practically? This is a legitimate worry, and due care must be taken when selecting benchmark tasks. Of course, it is inevitable that tasks judged inappropriate by some will be included in the common set, but each research group is free to choose those benchmark tasks that address issues of concern to them. If the set of common benchmark tasks does not provide a task of interest, then an appropriate task may be added. Community acceptance of a proposed task will come in the form of other research groups publishing performance results on that task.

An architecture can't be evaluated by a single benchmark. We do not propose to directly evaluate architectures through performance on an individual benchmark problem, but instead propose to evaluate

the performance of a specific system that is an instance of an architecture. Formal evaluation of the process by which an architecture is used to produce a solution to a specific benchmark problem is beyond the scope of this first effort. An architecture might include, but would not be limited to: design principles, programming languages, programming lore, user's manuals, existing code, etc. We cannot directly control for these architecture-related features, so the true worth of any given architecture cannot be directly evaluated by the method of benchmarks and metrics. Instead, an architecture will be evaluated in a "second-order" way: consistent success by the users of an architecture in applying that architecture to a widely-ranging set of benchmark problems will indicate the architecture's general utility.

4 Benchmark Tasks and Evaluation Metrics

4.1 Possible Task Attributes

Benchmark tasks can seen as varying across a number of dimensions. Placement in the space of possible task attributes will, in some sense, determine "task difficulty". As tasks are added to the evolving set of benchmarks, the importance of particular task attributes will become apparent.

It would be practically useful to have some families of tasks in which stress on selected task attributes could be systematically varied. This would allow researchers to experiment with a set of problems of graded difficulty and perhaps allow insight into how agent performance scales with task difficulty.

The following is an initial subset of possible task attributes.

Resource Management. Does the task have properties pertaining to metric time and continuous quantities? If so, the problem may take on the character of a classical optimization problem.

Geometric and Temporal Reasoning. Does the task involve extensive geometry or reasoning about activities over time? These kinds of reasoning may require specialized representations.

Deadlines. Does the task impose absolute deadlines for goal satisfaction, or does the utility of goal satisfaction vary continuously with time?

Opportunity for Learning. Is the task specification complete at the beginning of the agent's execution? If not, the agent must be able to acquire knowledge about its environment.

Multiple Agency. Does the task require the definition of a community of interacting agents? Such tasks might require complex communication protocols or reasoning about the internal states of other agents.

Informability. Can the agent be presented with explicit goals and facts about the world during the course of its execution?

Dynamic Environment. Does the agent's world change over time independent of the actions the agent takes? How predictable are the dynamics of the world?

Amount of Knowledge. How much *a priori* knowledge is available to be used by the agent? Some domains, such as medical diagnosis, require the assimilation of a large amount of domain knowledge.

Reliability of Sensors and Effectors.
In some task domains, sensors and effectors are completely reliable; in others, the main difficulty lies in accurately integrating data from a number of highly unreliable sensors and achieving robust overall behavior through the use of unreliable effectors.

4.2 Task Specifications

There are a broad range of possible methods for specifying tasks, ranging from informal to physical. Each of these methods has associated pros and cons; for instance, natural language task descriptions, simulators, and formal specifications are all easily transmitted electronically, but physical environments are less easily duplicated.

Natural Language Task Descriptions.
Natural language descriptions can be too vague for use in focused, comparative studies, but they are easy to generate, which makes them useful in certain situations. An example might be "Pick up cups using a simple mobile robot with a manipulator and an overhead vision system."

Simulators. Simulators provide a precise computational task description, which facilitates direct comparison between solutions. In addition, it is often possible to instrument a simulation to simplify debugging and evaluation.

Formal Specifications. Tasks may specified using a logical or mathematical description of the environment, its sensor and effector interfaces with the agent, and the goals of the agent. Formal specifications may allow prior analysis and provide insight into the underlying problem complexity. It is very difficult, however, to specify formally the majority of complex tasks.

Physical Environments. Some tasks are most easily specified by providing a physically embodied environment. Such specifications have the disadvantage of being difficult to replicate, because it is hard to control all aspects of the environment, such as lighting and RF interference.

A useful suite of benchmarks might include a variety of specification types for a particular general task; the resulting specifications would describe slightly different but closely related tasks. A useful example would be to have both simulation and physical specifications of a robotic control task. This would allow researchers to debug their ideas in simulation before trying them out in the real world. Although success in a simulation

is no guarantee of success in the real world, failure on a simulation will often entail real-world failure as well.

4.3 Modes of Evaluation

Part of the specification of benchmark tasks is the selection of particular evaluation metrics. Suitable metrics will become more obvious as the work progresses—some metrics will be applicable to all benchmark tasks and others will be task-specific. At this early stage however, two classes of metrics are of clear importance.

Agent Performance. How well does the agent perform? Performance will be measured using domain-specific performance metrics that are supplied in conjunction with each benchmark task.

Agent Construction. An important aspect of an integrated agent architecture is the ease with which it can be applied to a range of tasks. While it is difficult to formally measure the time to build a system, it can be informally reported in conjunction with agent performance results.

5 Conclusions

The development of a set of benchmark tasks and performance metrics for integrated agent architectures will foster both scientific progress and technology transfer. Broad coverage of the space of task attributes by the benchmark tasks will be required in order to ensure scientific and practical relevance of the benchmarks.

It is important to understand that no single research group will be able to determine a community-wide set of benchmarks by fiat. For a set of benchmarks to be accepted by a community, they must be designed by that community. The process presented in this paper, namely, that of a developing set of benchmarks that can be augmented by anyone willing to define a benchmark task, is one way to achieve such acceptance. The resulting set of benchmark tasks should be viewed as a resource that will foster progress, not as an exam for members of the community to pass or fail.

Acknowledgments

References

[1] K. Currie & A. Tate, 1985. O-Plan: Control in the open planning architecture. In *Proceedings of BCS Expert Systems '85*. Warwick, U.K. Cambridge University Press. pp. 225-240.

[2] M. Drummond, 1989. "Situated Control Rules," *Proceedings of the Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Canada.

[3] M. Drummond and J. Bresina, 1990. "Anytime Synthetic Projection: Maximizing the Probability of Goal Satisfaction," *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, Massachusetts, pp. 138-144.

[4] N. Garmonsway, 1965. *The Penguin English Dictionary*. Penguin Books Ltd, Harmondsworth, Middlesex, England.

[5] M. Georgeff and A. Lansky, 1985. "A Procedural Logic", *Proceedings IJCAI-85*, Los Angeles, Calif.

[6] L.P. Kaelbling, 1987a. "An Architecture for Intelligent Reactive Systems," *Reasoning About Actions and Plans*, M. Georgeff and A. Lansky, Eds., Morgan Kaufmann

[7] L.P. Kaelbling, 1987b. "Rex: A Symbolic Language for the Design and Parallel Implementation of Embedded Systems," *Proceedings of AIAA Conference on Computers in Aerospace*, Wakefield, Massachusetts.

[8] L.P. Kaelbling, 1988. "Goals as Parallel Program Specifications" Proceedings of the Seventh National Conference on Artificial Intelligence, St. Paul, Minnesota.

[9] J.E. Laird and P.S. Rosenbloom, 1990. "Integrating Execution, Planning, and Learning in Soar for External Environments," *Proceedings of Eighth National Conference on Artificial Intelligence*, pp. 1022-1029.

[10] N.J. Nilsson, R. Moore, and M. Torrance, 1990. *ACTNET: An Action Network Language and its Interpreter*. Draft paper, Stanford Computer Science Department.

[11] S.J. Rosenschein, & L.P. Kaelbling, 1986. "The Synthesis of Digital Machines with Provable Epistemic Properties," *Proceedings of Workshop on Theoretical Aspects of Knowledge*, Monterey, CA (March 13-14).

[12] S.J. Rosenschein & L.P. Kaelbling, 1989. "Integrating Planning and Reactive Control", *Proceedings of NASA Telerobotics Conference*, Pasadena CA.

[13] M.J. Schoppers, 1987. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proceedings of IJCAI-87*. Milan, Italy. pp. 1039-1046.

Collected Notes from
The Benchmarks and Metrics Workshop[1]
(NASA Ames, June 25, 1990)

edited by

Mark E. Drummond
Sterling Federal Systems
NASA Ames Research Center
Mail Stop: 244-17
Moffett Field, CA 94035

Leslie P. Kaelbling
Stanley J. Rosenschein
Teleos Research
576 Middlefield Road
Palo Alto, CA 94301

March, 1991

Contents

Introduction and Overview

An *integrated agent architecture* is a theory or paradigm by which one may design and program intelligent agents. An intelligent agent is a collection of sensors, computers, and effectors, structured in such a way that the sensors can measure conditions in the world, the computers can process the sensor information, and the effectors can take action in the world. Changes in the world realized by the effectors close the loop to the agent's sensors, necessitating further sensing, computation, and action by the agent.

In recent years there has been a proliferation of proposals in the AI literature for integrated agent architectures. Each architecture offers an approach to the general problem of constructing an integrated agent. Unfortunately, the ways in which one architecture might be considered better than another are not always clear.

There has been a growing realization that many of the positive and negative aspects of an architecture become apparent only when experimental evaluation is performed and that to progress as a discipline, we must develop rigorous experimental methods. In addition to the intrinsic intellectual interest of experimentation, rigorous performance evaluation of systems is also a crucial practical concern to our research sponsors. DARPA, NASA, and AFOSR (among others) are all actively searching for better ways of experimentally evaluating alternative approaches to building intelligent agents.

One tool for experimental evaluation involves testing systems on benchmark tasks in order to assess their relative performance. As part of a joint DARPA- and NASA-funded project, NASA-Ames and Teleos Research are carrying out a research effort to establish a set of benchmark tasks and evaluation metrics by which the performance of agent architectures may be determined. As part of this project, we held a workshop on *Benchmarks and Metrics* at the NASA Ames Research Center on June 25, 1990. The objective of the workshop was to foster early discussion on this important topic. We did not achieve a consensus of opinion, nor did we expect to.

This report collects together in one place some of the information that was exchanged at the workshop. This report includes an outline of the workshop, a list of the participants, notes taken on the white-board during open discussions, position papers/notes from some participants, and copies of slides used in the presentations.

Acknowledgements

The editors would like to thank Mel Montemerlo (NASA HQ), Lt Col Stephen E. Cross (DARPA), and Peter Friedland (NASA Ames) for making the workshop possible. Also, many thanks to Martha Del Alto for helping with the production of this report.

Workshop Outline

8:30 Coffee

9:00 Introduction

- General points.
- Domains and architectures of interest.
- Focus on characteristics of run-time system (*e.g.*, response time) vs. characteristics of development process (*e.g.*, ease of development.)
- Focus on external characteristics of system (*e.g.*, response time) vs. internal characteristics which may be architecture-specific (*e.g.*, number of subgoals generated.)
- How to (informally?) control for differences in underlying languages, compilation/interpretation environments, and machines.

9:30 How to specify a "benchmark task"

- What counts as a "task"? As a "metric"?
- Tasks and metrics must be chosen together.
- To specify task, need:
  * Environment description
  * Description of inputs & outputs of agent
  * Metric (Criteria for evaluating performance of agent)
- How should these be formulated (*e.g.*, formal, informal, via simulators, etc.)
- Compare and contrast agent benchmarks with standard benchmarks in other fields, *e.g.*, processor benchmarks, compiler benchmarks, etc.

10:30 Break

10:45 Three sample benchmark tasks/metrics and their attributes

- Presentation of strawman tasks in three domains:
  * *Tunable Benchmarks for Agent Evaluation* (M. Pollack)
  * *Benchmarks and Metrics for Mobile Agents* (R. Brooks)
  * *Requirements for Intelligent Monitoring Agents* (B. Hayes-Roth)
- Discussion will center on significant problem attributes exhibited by each of the strawman tasks (*e.g.*, time stress, uncertainty, run-time goals) and on objective metrics.

12:00 Lunch

1:00 Gap analysis

- Discussion will focus on the degree of match/mismatch between the tasks and metrics discussed in the morning and the specific tasks being used by workshop participants to evaluate their own work.
- The output will be a list of relevant task characteristics and a set of representative task instances embodying these characteristics, and possibly, some strategies and heuristics for generating and refining additional tasks and evaluation metrics.

3:15 Break

3:30 Standard robotic test platform

- Discussion of whether standard robotic platforms (*e.g.*, mobile robots, vision accelerators, etc.) should be made available to the research community and, if so, what their characteristics should be.
- How might the design of such standard platforms be established?
- How might platform development be funded?
- Who manages the distribution of the platforms?
- Is standardization premature? Will it stifle creativity?

5:00 End

6:00 Dinner

5

Workshop Participants

James Albus, NIST   albus@cme.nist.gov
Jim Antonisse, Mitre   antonisse@starbase.mitre.org
Hamid Berenji, NASA Ames   berenji@ptolemy.arc.nasa.gov
Pete Bonasso, Mitre   bonasso@starbase.mitre.org
John Bresina, NASA Ames   bresina@ptolemy.arc.nasa.gov
Rod Brooks, MIT   brooks@ai.mit.edu
David Chapman, Teleos Research   zvona@teleos.com
Steve Cross, DARPA   cross@vax.darpa.mil
Mark Drummond, NASA Ames   drummond@ptolemy.arc.nasa.gov
Jim Firby, JPL   firby@ai.jpl.nasa.gov
Carl Friedlander, ISX   cfriedla@isx.com
Armen Gabrielian, Thomson-CSF   tcipro!armen@unix.sri.com
Michael Genesereth, Stanford   mrg@sunburn.stanford.edu
Steve Hanks, U of Washington   hanks@cs.washington.edu
Barbara Hayes-Roth, Stanford   bhr@sumex-aim.stanford.edu
Jim Hendler, U Maryland   hendler@cs.umd.edu
Felix Ingrand, SRI   felix@ai.sri.com
Neil Jacobstein, Cimflex Teknowledge   njcaobst@teknowledge.com
Leslie Kaelbling, Teleos Research   leslie@teleos.com
Smadar Kedar, NASA Ames   kedar@ptolemy.arc.nasa.gov
John Laird, U of Michigan   laird@caen.engin.umich.edu
Pat Langley, NASA Ames   langley@ptolemy.arc.nasa.gov
Amy Lansky, NASA Ames   lansky@ptolemy.arc.nasa.gov
Rich Levinson, NASA Ames   rich@ptolemy.arc.nasa.gov
Dave Miller, JPL   dmiller@ai.jpl.nasa.gov
Steve Minton, NASA Ames   minton@ptolemy.arc.nasa.gov
Tom Mitchell, CMU   mitchell@daylily.learning.cs.cmu.edu
Andrew Philips, NASA Ames   abp@ptolemy.arc.nasa.gov
Martha Pollack, SRI   pollack@ai.sri.com
Paul Rosenbloom, ISI   rosenblo@venera.isi.edu
Stan Rosenschein, Teleos Research   stan@teleos.com
Marcel Schoppers, ADS   marcel@ads.com
Reid Simmons, CMU   reid.simmons@freesia.learning.cs.cmu.edu
Keith Swanson, NASA Ames   swanson@ptolemy.arc.nasa.gov
David Thompson, NASA Ames   det@ptolemy.arc.nasa.gov
David Tseng, Hughes   tseng@aic.hrl.hac.com
Dan Weld, U of Washington   weld@cs.washington.edu

White-Board Lists

This section contains three sets of items related to the evaluation of intelligent agent architectures:

(1) tasks that could be used to test proposed architectures;

(2) key attributes of tasks, or dimensions along which tasks can vary; and,

(3) requirements for hardware that could serve as a common experimental platform.

Lists (1) and (3) were generated by workshop participants in brainstorming sessions during the course of the workshop; list (2) was synthesized by the authors immediately after the workshop from ideas inspired by general discussion. In the spirit of the workshop, these lists are intended to be suggestive rather than comprehensive.

Possible Tasks

- Indoor robotic navigation and delivery.

- Outdoor survey and collection.

- Automated indoor tour guide.

- Multiple coordinated robots (with and without explicit communication).

- Satellite servicing (ORU replacement with verification).

- Find and assemble: definitions and instructions at runtime.

- Assemble, diagnose and repair electromechanical devices.

- Outdoor robots – real-time terrain reasoning.

- Cleaning and deburring machined parts.

- Space station assemply, maintenance, refueling.

- Process scheduling.

- Space shuttle refurbishment – scheduling, planning, diagnosis, repair.

- Cleaning teenager's room.

- Nurse's associate.

- Building construction.

- Data analysis planning (Earth Observing System).

- SIMNET player (robotic).

- Traffic world.

- Aircraft emergency procedures.

- Planning a tour in dynamic environment.

- Vehicle Driving.

- Rendezvous docking operations.

- Road grading.

- Beacon emplacement (to define landing areas).

- Structure protection (against external influences).

- Aerobrake assembly.

- Toxic waste handling and disposal.

- Ventilator management.

Possible Task Attributes

Resource Management. Does the task have properties pertaining to metric time and continuous quantities? If so, the problem may take on the character of a classical optimization problem.

Geometric and Temporal Reasoning. Does the task involve extensive geometry or reasoning about activities over time? These kinds of reasoning may require specialized representations.

Deadlines. Does the task impose absolute deadlines for goal satisfaction, or does the utility of goal satisfaction vary continuously with time?

Opportunity for Learning. Is the task specification complete at the beginning of the agent's execution? If not, the agent must be able to acquire knowledge about its environment.

Multiple Agency. Does the task require the definition of a community of interacting agents? Such tasks might require complex communication protocols or reasoning about the internal states of other agents.

Informability. Can the agent be presented with explicit goals and facts about the world during the course of its execution?

Dynamic Environment. Does the agent's world change over time independent of the actions the agent takes? How predictable are the dynamics of the world?

Amount of Knowledge. How much *a priori* knowledge is available to be used by the agent? Some domains, such as medical diagnosis, require the assimilation of a large amount of domain knowledge.

Reliability of Sensors and Effectors. In some task domains, sensors and effectors are completely reliable; in others, the main difficulty lies in accurately integrating data from a number of highly unreliable sensors and achieving robust overall behavior through the use of unreliable effectors.

Requirements for a Common Hardware Platform

- Manipulator.

- High speed data and control link.

- Easy to control from various machines.

- Payload capability.

- Modularity.

- Comes with a simulator.

- Standard bus card cage.

- Hierarchical sensor & effector software (felexible software library).

- Reliability.

- On-Platform debugging facility.

- Safety.

- Turnkey functionality.

- Affordable.

9

Position Papers and Notes

Some workshop participants contributed written position papers, and others provided short working notes. The following pages reproduce these papers and notes, with the authors' collective permission.

It is important to understand the status of the following material. The Ames meeting on June 25 was intended to be a *bona fide* workshop, and participants were encouraged to submit written (but preliminary!) accounts of ideas, issues, and concerns. To try to capture the spirit of the workshop, we have included some of this material in the following pages. Thus, these notes should not be read as a proceedings of the workshop, but rather, as a workshop "snapshot".

Please note that some authors have since expanded upon their workshop contributions. In particular, Paul Cohen has written a paper entitled "A Survey of the Eighth National Conference on Artificial Intelligence: Pulling Together or Pulling Apart" (AI Magazine, Spring 1991). Pat Langley, Leslie Kaelbling, and Mark Drummond have also written about related issues, and their papers can be found in the Proceedings of the DARPA workshop on innovative approaches to planning, scheduling and control, held in San Diego, CA, from November 5th through the 8th, 1990. The Proceedings are available from Morgan-Kaufmann, San Mateo, CA.

Authors and Titles

Jaime Carbonell, Tom Mitchell, & Allen Newell
The Diversity Metric for Intelligent Systems

David Chapman
On Choosing Domains for Agents

Paul Cohen & Adele Howe
Benchmarks Are Not Enough
Evaluation Metrics Depend on the Hypothesis

David Hart & Paul Cohen
Phoenix: A Testbed for Shared Planning Research

John Laird
Characteristics of Tasks for Intelligent Agent Benchmarks

Pat Langley
The Experimental Study of Intelligent Agents

Paul Rosenbloom
Informal Notes on the Nature of an Architecture

THE DIVERSITY METRIC FOR INTELLIGENT SYSTEMS
Proposed at the Benchmark and Metric Workshop, NASA Ames, 25 Jun 90
Revision #1, 18 Jul 90

Jaime Carbonell, Tom Mitchell, Allen Newell
School of Computer Science, Carnegie Mellon University

Preface: The forte of intelligent systems is the ability to deal with a broad diversity of tasks. We seek a measure of this ability to handle diversity.

Proposal:
> A **combo** is a **system** with a **developer**, where:
>> The **system** is the system to be measured.
>> The **developer** is a human with the expert skills and knowledge to operate, command, guide, instruct, reprogram and develop the **system** (hereafter, **modify the system**).

> A **task domain (D)** is a set of **tasks**, where:
>> Task accomplishment can be specified to some **competency level (CL)**.
>> The **domain** can be described coherently to a **developer**.

> The **completion time (T)** of a **combo** for a **task** from domain D is the time required by the **combo** to achieve the **competence level** specified for the task, where:
>> There can be prior knowledge of the domain D.
>> There is no prior knowledge of the **task** and **CL** before the clock starts.
>> The **developer** can **modify** the **system** at will.
>> The clock stops when the **system** performs the **task** by itself.

> The **diversity** of a system for a **domain D** is the set of **tasks** in D for which **completion times** are acceptable, where:
>> **Acceptability** is a parameter set to reflect the context of use in which the tasks occur.

Intelligent systems are to be evaluated in terms of the **diversity** they can achieve.

Notes:
1. The key idea is to permit development and modification, but to include this time in the measure of the system's effectiveness.

2. Modification time should not be separated from system-performance time, because the system's capabilities for flexibility may help as much in modification as in performance.

3. The system external interfaces (natural language, robotic, formal specification languages) play a critical role in the speed of its modification and the domains it can attempt. They are not fixed as part of the task specification, because the **combo** may modify them.

4. This metric assigns the full time cost of modification to a single performance, which is appropriate for many domains but not all; modification could be amortized over several performances.

5. Intelligent systems will be competitive for domains whose tasks pose significant requirements for assimilation, learning, adaptation and modification; specialized systems will be competitive for domains whose tasks are stable with prespecified variability.

6. The **combo** situation permits assessment before a system's adaptive capability is advanced enough to focus exclusively on measuring autonomous behavior.

7. The diversity metric is equally applicable to reuseability, maintainability and adaptability in software-engineering.

8. Interesting domains include: acquiring a new task domain, acquiring a major new functional capability, incrementing functionality, coping with a new environmental feature, accepting a new communication convention, exploiting a relaxation of time pressure, etc.

On choosing domains for agents

David Chapman

Teleos Research
576 Middlefield Road
Palo Alto, CA 94301
(415)328-8879   -

Abstract

Much recent work has addressed the problems of uncertain, real-time domains. This paper argues that these *negative* characterizations of domains are inadequate: they make an already difficult problem harder. Focussing on these characteristics leads one into intractable search problems and unrealistic noise models. We need rather to focus on the *positive* attributes of domains that make action possible. I analyze the success of Sonja and Pengi in these terms. I propose that research should seek new facilitating features and architectures that can exploit them.

1   Negative characterizations of domains

In our original paper on Pengi [1], Phil Agre and I characterized interesting domains as *complex, uncertain,* and *real time* and described Pengi's domain Pengo as one such. We focussed on these attributes of the domain to make the point that available theories of activity (based on planning) were unable to deal with important characteristics of real-world tasks.

In the past few years, uncertainty and real-time response have become central issues for theories of activity. (Complexity has been less studied; I'll come back to this.) This paper argues that while it is indeed important that real-world domains have these characteristics, focussing on them in isolation is a mistake.

1.1   Combinatorics

Planning problems are hard because of combinatorics. Adding uncertainty makes them much worse.

If all you know about a domain is that it is uncertain and real-time, all you can do is brute-force search. There's nothing grab onto to work with, no constraint to exploit. The search problems that come up in planning are unpleasantly intractable. What's worse, they aren't interesting. Without additional imposed structure, "planning is just computation" [2]; we can't expect to do anything clever.

Few researchers have tried to address the issue of domain complexity. The reason, I think, is that complexity is the feature that makes search infeasible. As long as activity research continues to focus on search-based techniques, the complexity of real domains must continue to be ignored.

Toy domains are, I believe, *too hard* because they are *too simple*. Complexity is a problem for search-based techniques, but it can be a resource for others. Simplifying a domain removes all the useful structure. (More about this later.)

The combinatorics of planning are the combinatorics of the problem, not of the solution. In other words, nontraditional solutions to the traditional planning problem can't work either. Since creatures do act sensibly in the real world, it must be that the *problem* is wrong, and too hard. It's wrong because it's too general. Neither people nor computers can act effectively in *arbitrary* uncertain, real-time domains.

1.2   Uncertainty, noise, randomness, and emergent structure

It's common these days to work in simulated domains in which certain aspects are subject to deliberately introduced pseudorandom variation. This is intended to model the observation

that real-world domains often vary unpredictably.

I believe that deliberately introducing simple forms of randomness is misleading and results in qualitatively different agent/world dynamics than the sorts of uncertainty that are prevalent in the real world.

Although the real world may contain some genuine randomness due to quantum noise, this is not the major source of uncertainty in the macroscopic domains. Uncertainty instead derives from lack of knowledge. The world is much too *complex* to represent all of it, even if you could find out about all of it, and too complex to simulate even if you could represent it. The world looks much more random, when viewed through the perspective of any one agent, than it actually is.

Why does this matter? It matters because beneath the apparent randomness of any particular real-world task there lies a great deal of *structure* which sensible agents rely on. The "noise" in real domains is not distributed uniformly, but in complex patterns which can—and must—be exploited to get useful work done. This exploitation may come from a bottom-up learning process or by explicit design by a theorist who also has observed and figured out how to make use of patterns whose causes are obscure.

By contrast, the noise that is added to simulated worlds is typically distributed uniformly or in some other simple way. This is because uncertainty is seen as a difficulty to be overcome, rather than a resource to exploit. If randomness is just an obstruction, it doesn't matter if it's just uniform. And indeed no use one can make of uniformly distributed noise; all you can do is work around it. All it does is make a bad scheduling problem worse. But if structured variation is a crucial resource for effective action, artificial noise results in an impoverished, unrealistically difficult world.

2   Positive characterizations of domains

In addition to explaining what makes the real world *hard* to act in, we need to say why it is *possible*. That is, we need positive characterizations of domains as well as negative ones. How can we choose facilitating characteristics in a principled way? I'll begin to answer this question by looking at some characteristics that successful existing systems exploit.

2.1   Why Sonja and Pengi work

Sonja [4] and Pengi [1] operate effectively in complex, uncertain, real-time domains (Amazon and Pengo). What features of these domains do they exploit?

It is impossible to answer this question definitely without extensive cross-domain research. I have some ideas, though. Sonja and Pengi work by means of visual routines that analyze the spatial configurations of the domain objects to extract task-relevant properties. For

example, Sonja looks to see whether it has a clear shot at a monster. What domain features enable such routines?

- In these domains, the effects of the agent's actions and of other processes are both spatially constrained. Amazons, ghosts, and fireballs move slowly enough that they traverse only small distances over the cycle time of the agent.

- Most of what is in any situation is irrelevant to any given task. Although any ice cube may become important at some point in a Pengo game, all but a few are uninteresting at any given time.

- Moreover, it is possible to find the relevant objects easily. This property along with the former two make it possible to avoid representing most of the domain, and thereby to avoid the combinatorics that the domain's complexity would otherwise engender. Finding relevant objects depends on other properties:

  - It is possible to perceive almost everything relevant with minimal effort. Sonja and Pengi have visual bandwidth limitations which model human visual bandwidth limitations, and these prevent the systems from looking at more than a tiny fraction of the visually presented scene at once. Moreover, in Amazon (but not Pengo) most domain objects are occluded at any given time. Nonetheless, it is usually possible in both domains to use visual search routines to find the most important objects in a situation quickly relative to the pace of the game.

  - Interesting objects are conspicuous (in terms of available perceptual primitives). For example, they may be moving, whereas uninteresting objects are mostly stationary; or they may distinctively colored, or distinctively placed in relationship to other interesting objects.

- Visually identifiable properties of objects strongly constrain their future behavior. Thus it is possible to reason about the future by means of geometrical computations (efficiently implemented in the visual system) rather than by deduction from frame axioms.

2.2   Future directions

The positive properties I enumerated in the last section are not true of every domain. However, they are mostly true of many domains most of the time. This makes them reasonable properties for an architecture to depend on.

More generally, interesting positive properties should not be domain-specific, or we will only learn about one domain. On the other hand, we should not expect to find positive

properties that hold of *every* domain of interest. Rather, we should enumerate many properties that are often true, and design architectures that can flexibly exploit those that are available and that degrade gracefully as facilitating features are removed.

There are several positive properties of domains that I suspect are true of most domains of interest, that I think important to human performance, and that I think we ought to investigate further.

- In many or most domains, activity is inextricably intertwined with perception. The sort of input an "action component" gets may be crucial to determining its architecture. Artificial domains that abstract away from perception may lead us down blind alleys. As I have argued elsewhere [3], I believe that the "simplification" of studying activity in isolation from perception makes the problem harder, not easier. The extraordinary richness of perception makes action easy.

- Real domains typically are regular or routine in the sense that a tiny subset of the logically or physically possible sorts of courses of events actually transpire. These regularities arise from complex interactions between domain processes. These processes and their interactions are typically unknown to agents acting in the domain, but in many cases are key to effective performance. How do agents come to exploit structure whose causes they don't understand?

- Typically people learn new tasks by trying the easy cases first. We need to understand how an agent can select its own training sequence and what properties of a domain make that easy. For example, a domain may make it perceptually simple to determine how difficult a problem instance is. A domain also has to make it possible for an agent to ignore or finesse the hard cases at first.

- Typically people learn new tasks by *legitimate peripheral participation* [5], i. e. by performing the task first as an assistant and gradually taking over more and more responsibility. What sorts of domains and tasks make legitimate peripheral participation possible, and how does this participation enable an agent to gradually take on more responsibility?

3 Summary

- Characterizing domains in solely negative terms leads to intractable and unrealistic pseudoproblems.

- We should, instead, look for the facilitating domain characteristics that make effective action possible.

- Such characteristics should not be domain-specific. but may also not hold of all domains all the time.

16

- Accordingly, no single domain can serve as an accurate benchmark. An architecture should be evaluated according to its versatility. We need a suite of benchmarks which are chosen to exhibit a range both of difficulties and resources.

- Domain complexity has been neglected as a source both of difficulty and opportunity.

Acknowledgements

References

[1] Philip E. Agre and David Chapman, "Pengi: An Implementation of a Theory of Activity," AAAI-87.

[2] David Chapman, "Planning for Conjunctive Goals." *Artificial Intelligence*, **32** (1987) pp. 333-377.

[3] David Chapman, "Penguins *Can* Make Cake." *AI Magazine*, Vol. 10, No. 4 (Winter 1989), pp. 45-50.

[4] David Chapman, *Vision, Instruction, and Action*. MIT AI TR 1204, April 1990.

[5] Jean Lave and Etienne Wenger, *Situated Learning: Legitimate Peripheral Participation*. Institute for Research on Learning Report No. IRL90-0013, February 1990.

Benchmarks Are Not Enough
Evaluation Metrics Depend on the Hypothesis

Paul R. Cohen and **Adele E. Howe**
Experimental Knowledge Systems Laboratory
Lederle Graduate Research Center
Department of Computer and Information Science
University of Massachusetts, Amherst

cohen@cs.umass.edu
howe@cs.umass.edu

June 1990

A couple of weeks ago we got an invitation: "to participate in the Benchmarks and Metrics Workshop (BMW), a one-day working meeting on evaluating agent architectures. The purpose of the meeting is to generate ideas on possible benchmark problems and evaluation metrics." Unable to attend, we console ourselves by writing this.

Our concern is that evaluation has become synonymous with performance evaluation. This leads to what John McCarthy is reputed to have called "look Ma, no hands" demonstrations. Benchmarks are a small step forward: they standardize what is being demonstrated, but they also perpetuate the view that evaluation is no more than demonstration of performance.

When we build AI systems as demonstrations, guided by benchmarks, we tend to aim for the benchmark and eliminate behaviors that, in the light of the benchmarks, we interpret as "bugs." As autonomous agents become more interesting, this style of research turns a blind (or prejudiced) eye to phenomena that lead to novel hypotheses. We end up building AI systems that do what they are expected to do. The literature is full of papers that assert, "We need X, here's how we expect to provide X, and (later) here's a demonstration of X." Lenat and Feigenbaum put it this way:

> "If one builds programs that cannot possibly surprise him/her, then one is using the computer either (a) as an engineering workhorse, or (b) as a fancy sort of word processor (to help articulate one's hypothesis), or, at worst, (c) as a (self-) deceptive device masquerading as an experiment. ... The most profitable way to investigate AI is to embody our hypotheses in programs, and gather data by running the programs. ... Progress depends on the experiments being able to falsify our hypotheses."

Hypotheses are already very rare in AI. The general form of a hypothesis is "X is sufficient to produce Y." But unlike hypothetico-deductive science, we rarely show the necessity of one hypothesis vis a vis mutually exclusive alternatives. Our principal mode is to accept the null hypothesis: to accrue demonstrations. It is difficult to see how instituting benchmarks—with their emphasis on demonstrating performance—will change this methodological stance.

Let us accept that AI research should be driven not by benchmarks but by hypotheses about why AI systems behave as they do in particular environments. What does this imply about evaluation metrics? There are two kinds of metrics: measures of performance, and measures of those factors we hypothesize are causally responsible for levels of performance (often called dependent and independent measures, respectively). Experiments test these causal hypotheses. In the discussions at this workshop, it seems essential to focus on a methodology to get at these causal hypotheses—on dependent and independent measures, both. Benchmarks *might* be designed to anticipate causal hypotheses, but failing that, they certainly should not be accepted in lieu of hypotheses.

Paul R. Cohen and Adele Howe. 1988. Toward AI Research Methodology: Three Case Studies. *IEEE Transactions on Systems, Man and Cybernetics.* Vol. 19, No. 3. pp. 634–646.

Paul R. Cohen and Adele E. Howe. 1988. How Evaluation Guides AI Research. *AI Magazine*, Winter, 1988. Vol. 9, No. 4, pp. 35–43.

Paul. R. Cohen. Evaluation and Case-based Reasoning. *Proceedings of the Second Annual Workshop on Case-based Reasoning*, Pensacola Beach, FL. May 30–June 2, 1989. pp. 168–172.

Lenat, D. and Feigenbaum, E.A., On the thresholds of knowledge. *Proceedings of IJCAI 10.* pp. 1173 -1182.

Phoenix: A Testbed for Shared Planning Research[2]

David M. Hart and **Paul R. Cohen**
Experimental Knowledge Systems Laboratory
Computer and Information Science
University of Massachusetts
Amherst, MA 01003

dhart@cs.umass.edu
cohen@cs.umass.edu

June 1990

Abstract

We describe an instrumented simulation testbed called Phoenix that we have developed for a complex, dynamic environment—forest fire-fighting. Phoenix has many built-in features to support the evaluation of autonomous planning agents. Thus, the Phoenix testbed bears consideration as a paradigmatic environment for current planning research.

1   Introduction

We have seen a revival in planning research in the last few years as attention has turned to AI systems that operate in complex, dynamic environments. A burgeoning number of task domains and a variety of new and old planning approaches are being explored. Several questions arise from such diversity: how do we evaluate this work? can we compare one approach to another? what characteristics do we require in a task environment, and which environments have those characteristics? In this note we describe an instrumented simulation testbed called Phoenix that bears consideration as one of a set of paradigmatic problems in current planning research.

Several years ago we chose a task domain that has many of the characteristics we associate with complex, dynamic environments—forest fire-fighting. Our goal has been to design autonomous agents for this environment. Our methodology requires empirical analysis of the environment and of the behaviors of the agents we design. To support this analysis, we built the Phoenix testbed [1] and a variety of tools for development and experimentation. Sections 2 and 3 describe the Phoenix task domain and the characteristics that make it paradigmatic. Section 4 describes the features of the testbed that support development and evaluation. Section 5 shows the layered modularity of the system and how other researchers could use part or all of it to test their own approaches to planning. Section 6 gives a partial list of current and potential areas of research in Phoenix to illustrate the richness of this task domain.

2   The Task Domain: Controlling Simulated Forest Fires

The Phoenix task is to control simulated forest fires by deploying simulated bulldozers, helicopters, fuel carriers, and other objects. The Phoenix environment simulates fires in Yellowstone National Park, for which we have constructed a representation from Defense Mapping Agency data. As the simulation runs, the user views fires spreading and agents moving through a topographical map of the park that shows elevations, ground cover, static features such as roads and rivers, and dynamic features such as fireline.

Fires spread in irregular shapes, at variable rates, determined by ground cover, elevation, moisture content, wind speed and direction, and natural boundaries. For example, fires spread more quickly in brush than in mature forest, are pushed in the direction of the wind and uphill, burn dry fuel more readily, and so on. These conditions also determine the probability that the fire will jump fireline and natural boundaries.

Fires are fought by removing one or more of the things that keep them burning: fuel, heat, and air. Cutting fireline removes fuel. Dropping water and flame retardant removes heat and air, respectively. In major forest fires, controlled backfires are set to burn areas in

21

the path of wildfires and thus deny them fuel. Huge "project" fires, like those in Yellowstone several years ago, are managed by many geographically dispersed firebosses and hundreds of firefighters. The current Phoenix planner is a bit more modest. One fireboss directs a number of bulldozers to cut line near the fire boundary under moderate conditions, or at some distance from the fire when it is spreading quickly.

3   Characteristics of the Fire-fighting Domain

Several characteristics of this environment constrain the design of agents, and the behaviors agents must display to succeed at their tasks. The fire environment is dynamic because everything changes: wind speed and direction, humidity, fuel type, the size and intensity of the fire, the availability and position of fire-fighting objects, the quantity and quality of information about the fire, and so on. The environment is ongoing in the sense that there isn't a single, well-defined problem to be solved, after which the system quits; but, rather, a continuous flow of problems, most of which were unanticipated. The environment is real-time in the sense that the fire "sets the pace" to which the agent must adapt. The agent's actions, including thinking, take time, and during that time, the environment is changing. Additionally, agents must be able to perceive changes in the environment, either directly through their own senses or indirectly through communication with other agents.

The environment is unpredictable because fires may erupt at any time and any place, because weather conditions can change abruptly, and because agents may encounter unexpected terrain, or fire, or other agents as they carry out plans. An agent must respond to unexpected outcomes of its own actions (including the actions taking more or less time than expected) and to changes in the state of the world.

The fact that events happen at different scales in the Phoenix environment has profound consequences for agent design. Temporal scales range from seconds to days, spatial scales from meters to kilometers. Agents' planning activities also take place at disparate scales; for example, a bulldozer agent must react quickly enough to follow a road without straying due to momentary inattention, and must also plan several hours of fire- fighting activity, and must do both within the time constraints imposed by the environment. (Notably, Phoenix agents are hybrid reactive/deliberative planners.)

The Phoenix environment is spatially distributed, and individual agents have only limited, local knowledge of the environment. Moreover, most fires are too big for a single agent to control. These constraints dictate multi-agent, distributed solutions to planning problems.

4 The Phoenix Testbed

The Phoenix testbed simulates the forest fire environment. It uses map structures to represent the forest, and a discrete event simulator to coordinate tasks that effect spreading fires, changes in the weather, and the actions of agents. The agents we design interact with this simulated world through a prescribed interface for sensors and effectors. Agent design and planning research are done in layers of the system built on top of the testbed (see Section 5).

The advantages of simulated environments are that they can be instrumented and controlled, and provide variety–all essential characteristics for experimental research. The Phoenix testbed has many features that facilitate experiments. It provides development tools for implementors, instrumentation for evaluation and analysis, baselines and baseline scenarios for benchmarking, and an interface for managing large experiments.

4.1 Development tools

The testbed includes tools to help developers implement and debug fire-fighting agents. The graphic interface (the map of the park) is highly interactive, allowing the user to zoom in and out at different resolutions, watch agents' movements and fire-fighting activities, determine static and dynamic features at any point in the map, and "see" the view of each individual agent (each agent's knowledge about dynamic features in the environment is determined by what it has perceived directly and what has been communicated from other agents). A desktop interface allows the user to interrogate each agent's memory structures, which are built from a generic frame system. A grapher and frame-editor allow the user to browse through memory structures and alter values for debugging.

4.2 Instrumentation

Phoenix is instrumented at three levels: the implementation level of system performance, the solution level of planning and agent designs, and the domain level of fire-fighting. Examples follow:

The implementation level includes metering to measure run time, cpu time , disk wait time, time since last run , idle time, and utilization, each graphed against time; as well as an interface to the Explorer performance metering that gives, for each function, the number of calls , average run time, total run time, real time, memory allocation, page faults, and so on.

The solution level is characterized by statistics on cpu utilization by the cognitive component of each agent, showing the profile of real-time response; for example, the latency between when actions on the timeline become available for execution and when they are

executed. Other metering provides metrics on sensor and effector usage, and on reflexes.

The domain level measures performance in domain terms; for example, the amount and type of forest burned, the number of houses burned, and the number of agents that perish. Resource allocation is currently measured by the amount and type of agents employed to fight the fire, gasoline consumed, fireline cut, distance traveled, and time required to contain the fire.

4.3 Baselines and baseline scenarios

We have developed baseline scenarios for several situations that are both characteristic of this domain and require timely response to environmental changes. As an example, one scenario involves a single fire whose profile changes dramatically due to an increase in wind speed and a change in wind direction, so that the nature of the threat changes from the potential loss of forest to loss of populated areas. Another scenario presents the problem solver with multiple fires and limited fire-fighting resources which must be managed efficiently to prevent one or more fires from spreading out of control. By limiting the number of fire-fighting agents available, and limiting the amount of fuel each can carry (requiring them to refuel periodically), this scenario forces the problem solver to allocate its resources wisely in order to control all the fires.

We are collecting a wide range of baseline data about the simulation environment that we will use to build our agents' planning knowledge- bases. This includes data about fire spread rates, agent movement, reaction and thinking time in agents, and effectiveness of fire-fighting strategies. Baseline data are also used to evaluate the performance of our agents.

A scripting capability allows the user to create and store scenarios. Scripts give control over environmental factors such as when and where fires start, wind characteristics, and the resources available for fire- fighting (how many agents of each type, what are their speeds, fuel capacities, fields of view). The timing of environmental changes is specified in scripts, allowing the user to control when events occur in the simulation for testing purposes. Instrumentation functions can be run within scripts.

4.4 Experimental interface

Scripts are part of a suite of data collection and analysis programs, which also includes facilities for manipulating data, a statistical package, and a graphing package. With these tools we have designed, executed, and analyzed the data from several large experiments involving—to date—over two thousand fire-fighting episodes.

5   Phoenix System Components

Phoenix runs on TI Explorers (color or B&W) and MicroExplorers, and is packaged into one system that includes all non-standard (non-proprietary) support code we use to run it. Part or all of the system can be provided to other laboratories and research groups on tape as a TI Load Band along with supporting documentation for the testbed, on-line help facilities, and annotated Lisp source code.

The Phoenix system comprises five levels of software:

1. DES — the discrete event simulator kernel. This handles the low-level scheduling of agent and environment processes. Agent processes include sensors, effectors, reflexes, and a variety of cognitive actions. Environment processes include fire, wind, and weather. The DES provides an illusion of simultaneity for multiple agents and multiple fires.

2. Map —this level contains the data structures that represent the current state of the world as perceived by agents, as well as "the world as it really is." Color graphics representations of the world are generated from these data structures.

3. Basic agent architecture— a "skeleton" architecture from which agents, such as bulldozers, helicopters, and firebosses are created. The agent architecture provides for sensors, effectors, reflexes, and a variety of styles of planning.

4. Phoenix agents— the agents we have designed (and are designing) for our own experiments.

5. Phoenix organization—currently we have a hierarchical organization of Phoenix agents, in which one fireboss directs (but does not control) multiple agents. Each Phoenix agent is autonomous and interprets the fireboss's directions in its local context, while the fireboss maintains a global view.

Phoenix is modularized so that other researchers can work with all or part of it. The first two levels, above, comprise the fire simulation testbed; researchers interested in designing and implementing their own agent architectures could experiment with them in this testbed. The next level is a generic agent architecture shell–a set of functional components common to all agents. Code to interface these components with the testbed is included with this level, so that researchers interested in working with our functional decomposition of agent capabilities need only instantiate the shells for components (sensors, effectors, reflexes, and cognitive capabilities) with their own designs. The fourth level includes our designs of these components, providing a specific agent architecture that is distinctive primarily for the planning style used in the cognitive component (skeletal planning with delayed commitment to specific actions). Researchers interested in using our planning style and agent architecture

could work with the first four levels, creating their own agent types and organizing them according to their research interests. The fifth level is the organization of fire fighting agents. Researchers interested in working with our solution to problem solving in this domain could use all five levels to replicate and/or extend our work

6   Research Issues in Phoenix

The Phoenix environment presents a rich variety of research issues. We are just scratching the surface with the ones we've tackled, which include: modeling the environment and agent architecture [3], building adaptive capabilities into the planner based on error recovery [6,7], sophisticated monitoring and control of plan execution [4], and real-time problem-solving [2,5]. Future areas of research we have identified are resource management, situation assessment, different protocols for the integration of reactive and deliberative control, and different types of learning.

7   Conclusion

Phoenix is an instrumented testbed that simulates a complex, dynamic environment. The task domain, forest fire-fighting, has many of the characteristics that define these environments. The testbed has many features designed to support empirical analysis, and is modularized for use by other researchers. This environment presents a variety of open research questions.

References

1. Paul R. Cohen, Michael L. Greenberg, David M. Hart, and A.E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3): 32-48.

2. Paul R. Cohen, A.E. Howe, and David M. Hart. Intelligent real-time problem solving: Issues and examples. *Intelligent Real-Time Problem Solving: Workshop Report*, edited by Lee D. Erman, Santa Cruz, CA, November 8-9, 1989, pgs IX-1–IX-34.

3. Paul R. Cohen. Designing and analyzing the Phoenix planner with models of the interactions between Phoenix agents and the Phoenix environment. Technical Report #90-22, Dept. of Computer and Information Science, University of Massachusetts, Amherst, March, 1990.

4. David M. Hart, Scott D. Anderson, and Paul R. Cohen. Envelopes as a vehicle for improving the efficiency of plan execution. Technical Report #90-21, Dept. of Computer and Information Science, University of Massachusetts, Amherst, March, 1990.

5. Adele E. Howe, David M. Hart, and Paul R. Cohen. Addressing real- time constraints in the design of autonomous agents. *Real-Time Systems*, 2(1/2): 81-97.

6. Adele E. Howe. Integrating adaptation with planning to improve behavior in unpredictable environments. In *Working Notes of the AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments*, Palo Alto, CA, March 1990.

7. Adele E. Howe and Paul R. Cohen. Responding to Environmental Change. Technical Report #90-23, Dept. of Computer and Information Science, University of Massachusetts, Amherst, March, 1990.

Characteristics of Tasks for Intelligent Agent Benchmarks

John E. Laird
Artificial Intelligence Laboratory
The University of Michigan
1101 Beal Ave.
Ann Arbor, MI 48109-2110
laird@caen.engin.umich.edu

June 22, 1990

1   Introduction

The purpose of this document is to lay out the space of task characteristics that should be covered by a suite of benchmark tasks for an intelligent agent. We assume that a task can be characterized by a task environment (such as a room with blocks), a set of goals (stack the blocks in the corner), and an agent consisting of a perceptual system (transducer of energy in the environment into information, such as a camera), a motor system (a transducer of information into energy/action in the environment, such as an arm), and a cognitive system (processor of information). The task environment may be real or simulated, with corresponding real or simulated perceptual and motor systems for the agent.

A benchmark would consist of a specification of task environment and possible constraints on the other components. For example, a benchmark might be just a task environment and a set of goals, where all of the other components are free variables that are under control of the system designer. Another possibility is that the benchmark would be the task environment, the goals, plus predefined perceptual and motor systems (such as those provided by a specific mobile robot). There could even be constraints on the cognitive system, such as that it must fit in a certain amount of memory or use only a prespecified set of domain knowledge.

2 Characteristics of the Environment

1. Dynamics of the environment

   (a) changes over time, independent of the agent

   (b) different external processes have different dynamics

   (c) synchronous vs. asynchronous with agent

   (d) predictable vs. unpredictable

2. Interaction between the environment and the agent

   (a) hinder vs. help the agent's goal achievement

   (b) recoverable vs. unrecoverable failures

   (c) tools can change interaction between agent environment

      i. provide transduction of environment to agent or vice versa.

      ii. can improve perceptual sensitivity, accuracy, precision, etc.

      iii. can improve action effectiveness, accuracy, precision, etc.

   (d) other agents may be in environment

      i. hinder agent goal achievement

      ii. help agent goal achievement

      iii. have own goals that are consistent or inconsistent with the agent

3 Characteristics of the Agent's Goals

1. multiple goals

2. interacting goals

3. concurrent goals

4. time dependent goals

5. one-shot vs. continuous vs. cyclic goals

6. hard vs. soft constraints on goal achievement

7. internal vs. external sources for goals

4   Characteristics of Perceptual System

1. sensitivity of sensor
   (which forms and ranges of energy)

2. completeness of sensor

3. amount of data from sensor

4. speed of sensor
   (timeliness of data)

5. precision of sensor

6. accuracy of sensor

7. active vs. passive sensor

8. fixed vs. manipulable sensor

5   Characteristics of Motor System

1. impact on environment
   (how effector changes environment)

2. physical range of effectors
   (no action at a distance)

3. degrees of freedom of effectors

4. concurrency of action
   (sequential vs. parallel action)

5. dynamics of effector
   (speed of action)

6. predictability of action

7. precision of action

8. accuracy of action

9. feedback of effect of action

6   Characteristics of Cognition

1. speed of processing

2. online vs. offline processing

3. memory limitations

4. access times for different memory structures

5. initial knowledge

6. correctness of initial knowledge

7. types of knowledge (episodic, procedural, semantic)

8. indirect sources of knowledge about environment
   (trainer, map, encyclopedia, data base)

9. ability to improve knowledge through experience

10. autonomy
    (can not be restarted by an external agent)

Acknowledgments

The Experimental Study of Intelligent Agents

Pat Langley
AI Research Branch, MS: 244-17
NASA Ames Research Center
Moffett Field, CA 94035

To evaluate any method or system, one needs some measure(s) of its behavior. In most experiments, these will be the dependent variables. One obvious measure concerns the quality of the agent's behavior, such as the length of solution path or the total energy expended. In some cases, the agent may be unable to accomplish a task or solve a problem, and one can also measure the percentage of solved problems. A third class of dependent variables involves the amount of time or effort spent in generating a plan or behavior (though in resource-limited situations, this may also be controlled).

Evaluation also acknowledges that there are different approaches to the same problem, and this constitutes a major independent variable in experimental studies. One can run different methods or systems on the same task and measure their relative behavior along one or more dependent dimensions. The goal here is not to hold a competition but to increase understanding. Nor need one compare entire systems against each other; one can also lesion specific components or vary parameters to determine their effects on behavior.

Seldom will one system always appear superior to another, and this leads naturally to the idea of identifying the conditions under which one approach performs better than another. Real-world problems may have practical import, but they provide little aid in factoring out the causes of performance differences. To accomplish this, one often needs artificial domains in which one can systematically vary domain characteristics. In tasks that involve planning and execution, some important characteristics include the complexity of the problem (e.g., the number of obstacles and length of the path in navigation tasks), the reliability of the domain (e.g., the probability that the agent's effectors will have the desired effect), and the rate of environmental change not due to the agent's actions. One can also view resource limitations (e.g., time or energy) as independent variables that affect task difficulty.

Different goals are appropriate for different stages of a developing experimental science. In the beginning, one might be satisfied with qualitative regularities that show one method as better than another under certain conditions, or that show one environmental factor as more devastating to a certain algorithm than another. Later, one would hope for experimental studies to suggest quantitative relations that can actually predict performance on unobserved situations. This should lead ultimately to theoretical analyses that explain such effects at a deeper level, preferably using average-case methods rather than worst-case assumptions.

However, even the earliest qualitative stages of an empirical science can strongly influence the direction of research, identifying promising methods and revealing important roadblocks. Research on planning and intelligent agents is just entering that first stage, but I believe the field will progress rapidly once it has started along the path of careful experimental evaluation.

Informal Notes on the Nature of an Architecture

Paul Rosenbloom
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292

1. It can be very difficult to come up with a firm evaluation of an architecture along particular dimensions. Occasionaly you find clear-cut cases where an architecture achieves optimal performance without requiring anything to be added or modified (either to the architecture itself, or on top of it [such as new knowledge]), or where an architecture is fundamentally incapable of exhibiting a certain form of behavior. However, most of the architectures are flexible enough to do just about anything if you put enough ingenuity into how it is used (this is the "Turing tarpit" problem). In such systems the tough questions come up in the large grey area between the two extremes. This is where questions like the following arise:

   - To what extent is the architecture implicated in the capability?

   - Does it provide the entire capability?

   - Does it provide support that could not be provided any other way?

   - Does it constrain how the capability is manifested?

   - Does the architecture say nothing about the capability?

   - Must an interpreter be built on top of the architecture in order to effectively provide the capability?

   - To what extent is the capability general across application domains?

   - To what extent does the capability integrate with the other requisite capabilities?

   - How much (and what type of) "user" effort is required to get the system to exhibit the capability?

Without answering such questions, data about which architectures can perform which benchmark tasks at what levels of performance are impossible to interpret. Even once you've answered these questions, there is still the question as to whether it is "appropriate" for the capability to be exhibited as it is – for example, certain types of behavior probably should be performed interpretively. We've written a paper in which we tried to provide preliminary answers to some of these questions for Soar, but it definitely isn't easy.

2. Ideally the set of benchmarks should not overstress any one particular capability over and above the others. What's needed is a set of benchmarks which stress different combinations (and numbers) of the basic capabilities. Of course, the real situation is even more complex that this, because "individual" capabilities – such as learning – really are a whole set of related, but distinct, capabilities. It is difficult (impossible?) to construct a set of benchmarks that doesn't bias the evaluation towards some specific capabilities, or even specific variations of the individual capabilities (such as explanation-based learning versus induction). If a level playing field cannot be achieved, there needs to be a discussion up front about the biases that lead it to slope in one direction or other.

Slides from the Presentations

We felt that presentations on specific tasks and metrics would help focus the discussion, so we invited three people well known for their research and experience with system evaluation: Martha Pollack, Rod Brooks, and Barbara Hayes-Roth. Dr. Pollack was expected to discuss experience gained with a simulated environment; Dr. Brooks was invited to discuss his significant experience with building real robot devices; and Dr. Hayes-Roth was asked to speak about her experience with the application of large knowledge-based systems to practical problems in monitoring and control. We felt that these three individuals represented an interesting range of applications and possible evaluation metrics. In this paper we do not attempt to provide a summary of the talks, but instead simply reproduce the speakers' slides (with permission) on the following pages.

Speakers and Topics

Rod Brooks
Benchmarks and Metrics for Mobile Agents

Martha Pollack
Tunable Benchmarks for Agent Evaluation

Barbara Hayes-Roth
Intelligent Monitoring: Environment, Behavior, Metrics

Benchmarks and Metrics for Mobile Agents

Rod Brooks
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139

Tunable Benchmarks for Agent Evaluation

Martha Pollack
Artificial Intelligence Center
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025

Requirements for Intelligent Monitoring Agents

Barbara Hayes-Roth
Knowledge Systems Laboratory
Stanford University

Appendix B: TCX

Section 6.2 described the use of TCX for software integration of Testbed API capabilities in a heterogeneous, distributed computing environment. The TCX software programming package developed at CMU provides for interprocess transfer of data structures while compensating for data format differences between varying machine and OS architectures. TCP/IP is the underlying protocol used by TCX to eventually communicate over a physical Ethernet network connection to other subsystems and processes. This Appendix contains documentation on TCX.

TCX

Task Communications

Christopher Fedor

1 February 1993

Version 7.7 (14-Jan-93)

1. Introduction

TCX - Task Communications is an interprocess communication package designed to support mobile robotic applications across multiple hardware platforms and programming languages. The goal of TCX is to provide routines for controlling the sending and receiving of information and maintaining channels of communication between cooperating processes. More specifically TCX provides the communication building blocks to coordinate flow of control and flow of data between planning, perception and real time control in mobile robots.

TCX is very much a product of its users. Each new application refines functionality and corrects errors. Much of TCX's success is due to its continued use in real robotic applications and the desire to make the system as easy to use as possible. The Erebus Software group has greatly improved the functionality and code of TCX and have suffered the most during its development. Members of the Erebus Software group are Christopher Fedor, David Wettergreen, Jay West, Paul Keller, Dan Christian, Bill Ross, Jim Bowlby, Chad Darby, Sean Goller and Christopher Stengel. Additional advice and work was provided by Hans Thomas, Jay Gowdy, Rich Goodwin, Bala Kumar, Chuck Thorpe, Tony Stentz and Reid Simmons.

Comments can be e-mailed to fedor@cs.cmu.edu

This version of the manual reflects what has been implemented with some notes on what should be changed. In general, future TCX versions will simplify message receive calls and allow more control over which handlers can be executed for a particular message. The data layer may be hidden or removed altogether.

TCX improves with use and your comments. This version is still considered preliminary with a new design and implementation in progress. Ideas you have found useful or not so useful can be emailed to fedor@cs.cmu.edu.

2 History / Design Notes

TCX grew out of the success of TCA, the Task Control Architecture which was created for the Carnegie Mellon Planetary Rover group. TCA provided simple semantics for interacting modules which TCX built on for a project which did not require TCA's integral support for planning. The design of TCX also takes into consideration the observations of robot system design from EDDIE. The EDDIE architectural toolkit was developed for the Carnegie Mellon Navlab program. Unlike TCA, EDDIE explicitly avoided robotic architectural issues choosing instead to provide only a common set of communication routines. TCX is a superset of EDDIE and attempts to bridge the gap between EDDIE and the communications layer of the Task Control Architecture.

2.1 Observations from EDDIE

• Centralization is a bad idea at low levels.

• Most communication patterns are fixed.

• Most control flow is fixed.

• Most state is local

2.2 Observation from TCA

• Central high level control need not be a bottleneck and simplifies debugging.

• Descriptive messages and data formats which are similar to the programming language being used speeds inter-process communication development time.

• Messages which tap or trigger on other messages simplifies optional modules and user interface additions. Message tapping can also lead to a sometimes confusing increase in number of messages.

2.3 Observations from TCX 0.0

• UNIX TCP/IP dominates inter-process communication transit time.

• A fixed mapping from messages to their data formats is desirable.

• The ability to receive messages without executing a handler simplifies sever style control.

2.4 Additional Observations

• A software toolkit does not prevent chaos in system design A communications toolkit may simplify system integration but it should not be seen as a magic cure for a bad design.

• Access to lower communication layers should be provided. There should be a common method for using both messages passing and byte passing in the same module.

• Point to Point communication is intuitive but "central" modules develop based on control flow or data flow

3 TCX Overview

TCX is an interprocess communications library which provides message passing semantics on top of unix interprocess communication routines. TCX provides interprocess communication routines which define messages to support data abstraction, handlers to support procedural abstraction and message receive routines to support control abstraction. TCX also supports to a limited extent methods for passing data without using the message system for special applications.

A session with TCX involves two or modules which wish to share information and a communications server. Modules define the messages they will be using and makes requests to connect to another module through the communication server. The communications server establishes a direct connection between sending and receiving modules and attempts to maintain those connections as modules exit and restart.

The sending module sends a message, which may include data, and a receiver picks up the message. The receiver can control if a procedure should be executed with the incoming message and data or if the message should be handled within a main loop. The receiver may also optionally reply to the sent message.

3.1 Communications Server & Communicating Modules

TCX coordinates module interprocess communications through a communications server. Communication is based on a notion of a module. The term module denotes a separate running process on some machine. When a module starts, it initializes a connection to the communications server and informs the server of those messages it wishes to send and receive. The communication server then establishes inter module connections and insures that only one bi-directional connection between any two modules is created.

3.2 Asymmetric Communication

An interprocess communication facility must supply some method for sending and receiving information. TCX provides an asymmetric method for send and receive calls. The communication is considered asymmetric because the sender must name the destination but the receiver can receive from any sender without naming them. Send and Receiving information is defined as:

• **Send***(module, message)*. Send a *message* to a *module* reference. Send calls are nonblocking.

• **Receive***(module, message, data)*. Receive a *message* and *data* from any module. Set *module* to the sender. Receive calls are blocking calls with an optional timeout. Receive also establishes new module connections.

The advantage of this method is that it allows for easy coding of synchronization and rendezvous between separately executing processes. A possible disadvantage of this method is the need to specify a specific module name. This can lead to recompiling if a module name changes or additional work to support sending the same information to multiple or optional modules. A fixed module name also complicates crash recovery. Some of these disadvantages are avoided by having a module pointer which refers to a module as opposed to a fixed string or fixed id value.

3.3 Indirect Communication

Another method for sending and receiving information is to not specify the receiver of the information. This is usually accomplished by sending information to an abstract mailbox and those modules which share mailboxs can communicate. This avoids communication complications with having to name modules explicitly to send information. Optional modules, such as simulators and user interfaces, which are not always part of a running system would simply pick up a message at a known point without the sender being flagged that an optional module is running.

Indirect communication in TCX is not yet implemented. Current design would have a sender simply send a message and those modules which are interested in receiving that message would receive it. This method is already implemented in TCA. Additional concerns for TCX involve questions of informing the sender of who received the message and should multiple receivers be allowed to execute the same message.

3.4 FIFO Message Queue

Each module maintains a FIFO message queue of incoming messages. When a receive call is made, each connection with pending information is read and the queue is updated. New connections are also established at this time. The receive call then checks to see if the request specified by the call can be filled. If it can the call returns with the information, if not the process is repeated until the receive request can be filled. Routines are provided to flush the message queue.

For modules which receive message from multiple modules, there is no guarantee of the module order of messages received. A module is guaranteed that messages received from a particular module are received in the order that module sent. A module can also send two messages so that they appear in the receive queue sequentially. In general, a module processes messages in the order they are received unless waiting for a special reply.

The message queue provides a temporary storage area which allows modules to select the order in which messages are received. It also allows a module to internally store incoming messages while waiting for a reply message from a query for data. The message queue operates transparently for a user of TCX. For most TCX modules the message queue appears as a stack processing messages as they are received. Future versions of TCX are looking at providing more user level routines for the message queue.

4 TCX Module Connections

TCX provides routines which initialize module connections, inform the communications server of a module's location and establish point to point bidirectional connections between modules. The term *module* in this document refers to a separate running process on a particular machine which makes use of the communications server. TCX also attempts to maintain connections among a collection of modules when a module crashes and reconnects.

4.1 Starting TCX Communication server

TCX files include the executable communications server, *tcx*, the library of TCX routines, *libtcx.a* and the include file of TCX routines, *tcx.h*. The communications server is started by issuing *tcx* at the shell prompt. When the communications server starts it will display version information as well as a release date. The server will also display information about module connections as they occur. The communications server does not have to be the first process started.

4.2 tcxInitialize

Initializes a connection to the communications server.

void tcxInitialize(char *moduleName, char *serverHostName)

Initializes TCX for module, *moduleName*, and connects the module to the communications server on machine, *serverHostName*. *tcxInitialize* should be called before any other TCX routines. The communication server need not be started first. If the module does not find the server on the host machine it will display a message and try again every 2 seconds until a connection is established.

Once a connection is established with the communications server version information is exchanged. Version information is coded as two integer values, a major version id and a minor version id. If the module's major version id differs from the communication server's major id value the module will exit. If the minor version id differs a warning message is displayed. If a warning message is displayed the system should still work but relinking with the current TCX *libtcx.a* is recommended.

tcxInitialize should be called before calling additional TCX routines. The current exception to this is the routine *tcxRegisterConnectHnd* which sets special handlers for modules which may connect during a *tcxInitialize* call.

4.3 tcxInitializeServerRoute

Not Yet Implemented - Initialize a module such that all messages get routed through the server

void tcxInitializeServerRoute(char *moduleName, char *serverHostName)

Initializes TCX for module, *moduleName*, and connects the module to the communications server on machine, *serverHostName*. *tcxInitializeServerRoute* is identical to tcxInitialize except that all messages being sent and received for this module are to be sent first to the communication server

and then on to the intended destination. This routine exists as a simple method to control the number of socket connections a module establishes without changing the semantics of send and receive TCX routines. A simpler, more flexible, method has been designed for use with TCX indirect message communication. This routine will probably go away with future releases.

4.4 tcxConnectModule

Connect a module to another module.

TCX_MODULE_PTR tcxConnectModule(char *moduleName)

Establishes a connection to the named module and return a reference to that module. *tcxConnectModule* is a blocking call.

4.5 tcxConnectOptional

Connect a pre-existing module.

TCX_MODULE_PTR tcxConnectOptional(char *moduleName)

If the named module has already initialized a connection to the communications server this call establishes a connection to the named module and returns a reference to that module. If the module does not exist NULL is returned.

4.6 tcxSetAutoReconnect

Flag a module for auto-reconnect through a TCX receive call.

void tcxSetAutoReconnect()

TCX modules written as servers typically call *tcxRecvLoop*, which receives a message and then calls a handler for that message. If a call to *tcxRecvLoop* is made the communication server is informed that it can assume that processing will eventually return to a TCX receive call. If auto reconnection is needed for this module the communication server can send a reconnect message to this module and be sure that the module will be listening for incoming messages.

Modules which only send information are not flagged for auto-reconnect messages because they will not be calling TCX receive calls and would only produce a dead lock.

Another method of writing a server module is to use *tcxReceiveMsg* in a loop and decide whether or not to call a handler to process the message. This is similar to creating a special *tcxRecvLoop*. The call *tcxSetAutoReconnect* sets the reconnect flag and informs the communication server that the module will be listening for incoming messages and it is ok to send reconnect messages to this module.

4.7 tcxModuleName

Given a module reference return a pointer to the string name.

char *tcxModuleName(TCX_MODULE_PTR module)

Returns a pointer to the name of a module. This should be byte copied if it should exist after the module exits.

4.8 tcxTestActiveModule

Test if a module pointer points to an active reference to a module.

int tcxTestActiveModule(TCX_MODULE_PTR module)

Returns 1 if the module reference, module, points to a module that is an active connection. Returns 0 otherwise.

4.9 tcxCloseAll

Closes all module connections currently active.

void tcxCloseAll();

All connections to a module are closed. All socket connections are closed and memory representing a module is freed. Global references to a module will become invalid after this call.

5 Describing Data

Data formats provide a method for describing the structure of a user data type so that data can move between separate processes in a transparent manner. The message layer of TCX provides routines for registering data formats which are used to encode the data into a linear stream of bytes, transfer the byte stream between processes, and reassemble the data at the receiving end. The use of data formats for transferring information between processes is optional.

TCX supports a wide class of different data formats. Primitives include data types such as integer, float, double and string as well as special data types for variable dimension arrays of floats, and for transferring parts of these arrays. TCX also provides the ability to compose primitive data types by forming structures, fixed length arrays, variable length arrays, and pointers to data formats.

5.1 Data Formats

The user represents format information as a string which is parsed by TCX into an internal format representation. The data format string reassembles a C language typedef and specifies the layout of a particular data structure. The data format string may be either a primitive type or a composite type. Primitive types included in the data format language are byte, char, short, int, float, double, boolean and string.

A boolean is defined in C as 0 for FALSE and 1 for TRUE. In Lisp a boolean translates to NIL, for FALSE and T for TRUE. The type string in C is treated as a NULL terminated ('\0') list of characters. Additional special primitives are included for matrix operations.

An example of a primitive data format is:

```
typedef int SAMPLE_INT_TYPE;

#define SAMPLE_INT_FORM "int"
```

Composite data formats are aggregates of other data types. The supported composites include structures, fixed length arrays, variable length arrays, pointers and self referencing pointers.

• Structures are denoted by a pair of braces surrounding a list of data types separated by commas:

```
typedef struct {
        char *str;
        int x;
        int y;
} SAMPLE_STRUCT_TYPE;

#define SAMPLE_STRUCT_FORM "{string, int, int}"
```

• Fixed Length Arrays are denoted by square brackets that enclose a data format specification, a colon, and then a list of one or more dimensions separated by commas:

```
typedef struct {
        int array[17][42];
        char *str;
} FIXED_ARRAY_TYPE;
```

```
#define FIXED_ARRAY_FORM "{[int: 17, 42], string}"
```

• Variable Length Arrays are denoted by angle brackets. The data format specification is the same as for fixed length arrays, except that the dimension numbers refer to an element of the enclosing structure that contains the value of the actual dimension. In the VARIABLE_ARRAY_FORM example, the "1" refers to the first element of the structure, which contains the number of elements in the variable length array. Variable Length Array formats must be directly embedded in a structure format:

```
typedef struct {
        int arrayLength;
        int variableArray[];
} VARIABLE_ARRAY_TYPE;
```

```
#define VARIABLE_ARRAY_FORM "{int, <int: 1>}"
```

• Pointers are denoted by an * followed by a data format specification. If the pointer value is NULL (or NIL in Lisp) no data is encoded or sent. Otherwise data is sent and the receiving end creates a pointer to the data. Note that only the data is passed, not the actual pointers, so that structures that share structure or point to themselves (cyclic or doubly linked lists) will not be correctly reconstructed.

```
typedef struct {
        int x, *pointerToInt;
} POINTER_EXAMPLE_TYPE;
```

```
#define POINTER_EXAMPLE_FORM "{int, *int}"
```

• The self pointer definition, *!, is used for defining linked or recursive data formats. The self pointer format refers to the enclosing data structure. TCX will translate linked data structures into a linear form before sending them and then recreate the linked form in the receiving module. These routines assume that the end of the linked list is designated by a NULL pointer value. Therefore it is important that all linked data structures be NULL for the encoding routines to work correctly.

```
typedef struct _EXAMPLE {
        int x;
        struct _EXAMPLE *next;
} EXAMPLE_TYPE;
```

```
#define EXAMPLE_FORM "{int, *!}"
```

In addition to specifying data as a list of defined primitives, one can also give an integer number specifying the number of bytes to send. In the examples below each data format is equivalent. *This option is not available in lisp.*

```
typedef struct {
        int x;
        int y;
        char *string;
} SAMPLE_TYPE;

#define SAMPLE_DATA_FORM_1 {int, int, string}"

#define SAMPLE_DATA_FORM_2 "{4, 4, string}"

#define SAMPLE_DATA_FORM_3 "{8, string}"
```

6 Message Layer

There are two interfaces for sending information from one module to another. The message layer interface provides string names for hiding the details of sending information to another module. When message layer routines are used a hash table lookup is performed on the message name to determine the details of sending information to another module. These details include an integer message id value, data formats to encode and decode information and optionally an integer reference value to tag this particular instance of a message. The message layer is built on a TCX data layer which makes these details much more explicit. Both message and data layer routines can be used in the same module. Both layers make use of the same message receive queue for receiving information from other modules.

Most of the details for setting up messages are handled by the communications server. In future TCX releases the message layer will be emphasized over the data layer. Some data layer routines will be removed.

6.1 Defining Messages

A TCX message definition provides a mapping from a string message name to a format for encoding and decoding data. A message is defined by registering an array of messages. Each element of the array is of type TCX_REG_MSG_TYPE. The array of messages is usually declared statically in an include file but the array only needs to have valid information during registration.

```
typedef struct {
        char *msgName;
        char *msgFormat;
} TCX_REG_MSG_TYPE;

typedef struct {
        char *name;
        char *format;
        int connections;
} TCX_REG2_MSG_TYPE;
```

TCX_REG2_MSG_TYPE is provided for future work with Indirect Communications.

A message definition includes the message name, *msgName*, a format for encoding/decoding data sent, *msgFormat*, and a message format for encoding/decoding data received as a response to some message routines, *resFormat*. NULL can be used as a legal value for msgFormat or resFormat.

```
TCX_REG_MSG_TYPE messageArray[] = {
        {"sampleMsg", "{int, {float, float}}"},
        {"posRequestMsg", "int"},
        {"posInfoMsg", "{float, float, float, float, float, float}"}
};
```

6.1.1 tcxRegisterMessages

Define a set of messages.

void tcxRegisterMessages(TCX_REG_MSG_TYPE *msgArray, int size)

Defines the mapping from a message to an encoding/decoding data formatter. A module will send its message array to the communication server and receive an integer id value for each message registered which will be stored internally and used to send a message. Each message name is unique but is not case sensitive.

A common method in C to determine the size of an array can be used to specify the *size* parameter. This example can also be defined as a macro in the include file defining the message array.

```
tcxRegisterMessages(messageArray,
        sizeof(messageArray) / sizeof(TCX_REG_MSG_TYPE));
```

6.1.2 tcxMessageName

Given a message id value return a pointer to the string name of the message.

char *tcxMessageName(int id)

Returns a pointer to a message name given an internal id value. If no name matches then NULL is returned. The id value is set by the call tcxRegisterMessages from the communications server. The id value is not guaranteed to be the same for multiple runs.

6.1.3 tcxMessageId

Given a message name return the internal id value.

int tcxMessageId(char *name)

Returns the id value of a given message. If no such message exists then the value 0 is returned. The id value is set by the call tcxRegisterMessages from the communications server. The id value is not guaranteed to be the same for multiple runs.

6.2 Sending Messages

6.2.1 Instance Ids

Some calls return or make use of a generated instance value. This is a non-zero integer value that identifies a specific instance of a message. The internally generated values simply increment an integer counter each time a message is sent. Because of this the id value may become negative and may repeat after a sufficient number of messages.

The instance value is used to determine which unique instance of a message was sent. The TCX call *tcxQuery* makes use of the instance value to map replies to the correct query message.

6.2.2 tcxSendMsg

Send a Message to a Module.

int tcxSendMsg(TCX_MODULE_PTR module, char *msgName, void *data)

tcxSendMsg sends a message and data to a specific module. The call returns an internally gener-
ated instance id value for this particular message.The format used to encode the data is found
from the message, *msgName. tcxSendMsg* is a non-blocking call. If an error occurs while sending
information or if the module becomes invalid the tcxSendMsg call may not be able to detect the
error until a future TCX call. See the section on error recovery.

6.2.3 tcxSendDoubleMsg

Send two messages at the same time to a module.

int tcxSendDoubleMsg(TCX_MODULE_PTR module, char *name1, void *data1, char *name2, void *data2)

tcxSendDoubleMsg combines two messages into a single send call. The messages are guaranteed
to be placed in the receiving queue in order with no other message separating them. The value
returned is the instance id value generated. The same instance value is used for each message. *tcx-
SendDoubleMsg* is a non-blocking call.

6.3 Receiving Messages

Receive message calls control how often and in what order messages which have been received
are decoded. Receive message calls also control whether or not user procedures are executed
when a messages is received. By default receive message calls are blocking calls which wait for a
particular message or at least a message handler to be executed. In many cases, a timeout parame-
ter is provided to effect a polling condition.

Receive message calls are based on and implemented with receive data calls and the internal data
formatting code. *Future TCX work will refine the receive message calls and possibly remove the
receive data calls.*

6.3.1 tcxRecvMsg

Receive a particular message while allowing user handler procedures to be executed.

int tcxRecvMsg(char *msgName, int *ins, void *data, void *timeout)

tcxRecvMsg waits to receive the message, *msgName*, and stores the data associated with the mes-
sage in data using the formatter with which the message was registered. An optional instance
value, *ins*, is used to either select a particular instance of *msgName* or to receive the instance value
of the received message. *tcxRecvMsg* returns the number of bytes in the encoded message or
returns -1 on timeout.

While *tcxRecvMsg* is waiting for a particular message to arrive, it will execute handlers for other messages that arrive before the message specified by *msgName*. These handlers are further discussed with the call *tcxRegisterHandlers*.

The most common form of *tcxRecvMsg* is:

```
tcxRecvMsg("sampleMsg", NULL, &sampleData, NULL)
```

Will wait until the message sampleMsg has been received and use format information to fill in the structure sampleData with the data from this message.

Other forms which wish to know the instance value of sampleMsg would use:

```
int instance;
instance = 0;
tcxRecvMsg("sampleMsg", &instance, &sampleData, NULL)
```

Which will receive sampleMsg and record the instance value of this message in instance.

To wait for a particular instance of a message one would use:

```
int instance;
instance = 17;
tcxRecvMsg("sampleMsg", &instance, &sampleData, NULL)
```

This would wait for sampleMsg whose instance id value is equal to 17.

A timeout value is provided to indicate how long the call should wait to receive this message. If timeout is NULL, the call is blocking and will wait for the message to be received. If timeout.-tv_sec = 0 and timeout.tv_tusec = 0, then the call effects a simple poll to check if the message has been received.

timeout is a pointer to struct timeval from #include <sys/time.h>

```
struct timeval {
        long tv_sec; /* seconds */
        long tv_usec; /* microseconds */
}


struct timeval timeout;
timeout.tv_sec = 1;
timeout.tv_usec = 0;
tcxRecvMsg("sampleMsg", NULL, &sampleData, &timeout)
```

Would wait 1 second to receive the message "sampleMsg", possibly executing one or more handlers for other messages already in the pending message queue. If "sampleMsg" is not found or received within 1 second, *tcxRecvMsg* will return with a -1 timeout value.

If msgName is NULL tcxRecvMsg simply returns 0 and does not execute any handlers or check for new messages.

In cases where one simply wishes to wait until any message is received and execute one or more handlers see *tcxRecvData*.

6.3.2 tcxRecvMsgNoHnd

Wait for a particular message without executing user defined message handlers.

int tcxRecvMsgNoHnd(char *msgName, int *ins, void *data, void *timeout)

tcxRecvMsgNoHnd is identical to *tcxRecvMsg* except that while waiting for a particular message no user handlers are allowed to run. Internal tcx message handlers, auto-reconnection is one example, will still execute.

6.3.3 tcxRecvMsgE

Wait for a particular message from a particular module.

int tcxRecvMsgE(TCX_MODULE_PTR *module, char *msgName, int *ins, void *data, int allowHnd, void *timeout)

tcxRecvMsgE provides most of the functionality of tcxRecvData. The module parameter controls if the message is to be received from a particular module or if the module which sent the message is to be identified.

If module field is initialized to NULL the sender will be returned. This example will store the module which sent "sampleMsg" in module.

```
TCX_MODULE_PTR module;
module = NULL;
tcxRecvMsgE(&module, "sampleMsg", NULL, &sampleData, ALL_HND, NULL)
```

Otherwise if a particular module is referenced then the message must be received from that module. In this example "sampleMsg" must be received from module b1.

```
TCX_MODULE_PTR module;
module = tcxConnectModule("b1");
tcxRecvMsgE(&module, "sampleMsg", NULL, &sampleData, ALL_HND, NULL)
```

If the module field is NULL then *tcxRecvMsgE*, behaves like *tcxRecvMsg* or *tcxRecvMsgNoHnd*. How tcxRecvMsgE deals with handlers is defined by the allowHnd field. The allowHnd field

should either be set to the defined value ALL_HND or TCX_HND. ALL_HND means that while waiting for this message. *Future TCX designs may expand and simplify control over handlers most applications should try to make use of tcxrecvMsg.*

6.3.4 tcxRecvLoop

Loop on receiving messages and executing handlers.

int tcxRecvLoop(void *timeout)

tcxRecvLoop is a loop which receives an incoming message and executes the handler associated with that message until a timeout is reached. The parameter *timeout* is of type struct timeval found in the standard include file sys/time.h. If a timeout is reached the value returned is -1. If the a message is received for which no handler exists an error message is displayed and the value -2 is returned. *tcxRecvLoop* simply loops on the call *tcxRecvData.*

The call *tcxRecvLoop* calls *tcxSetAutoReconnect* to inform the communication server that this module will be able to receive reconnection messages.

6.3.5 tcxFree

Free message data by message name.

void tcxFree(char *msgName, void *data);

tcxFree looks up the formatter information from *msgName* and frees memory pointed by *data.* This attempts to avoid memory fragmentation by using tcxFree to free message data in the same manner in which it was allocated. *tcxFree* easily handles freeing of complex message data.

6.4 Query / Reply

Asking for information from a module and receiving a reply is a common operation. A common problem is to insure that the reply is matched with the question which triggered it. If multiple mapRqstMSG are sent to a mapping module, the mapping module is free to answer the request in any order. The module which requested the information would need to insure which reply map went with which mapRqstMSG. This can be accomplished by using the instance value of messages. *tcxQuery* and *tcxReply* simplify this bookkeeping.

6.4.1 tcxQuery

Send a query and wait for its reply.

void tcxQuery(TCX_MODULE_PTR module, char *MsgName, void *data, char *replyMsgName, void *reply)

tcxQuery is a blocking call which implements a *tcxSendMsg* and *tcxRecvMsg* pair to send a message to a module and then waits until a reply for that instance of the sent message has been received. Since *tcxQuery* makes use of *tcxRecvMsg*, user handlers are active for other messages

while waiting for the reply.

6.4.2 tcxReply

Reply to a particular query.

void tcxReply(TCX_REF_PTR ref, char *replyMsgName, void *reply)

Replies to *tcxQuery* calls are generally from within handler procedures. *tcxReply* makes use of a handler's ref pointer to send a reply message and data to the sender with the correct instance value of the message which triggered this handler.

6.5 Message Queue Routines

6.5.1 tcxRecvFlush

Empty the receive message queue.

void tcxRecvFlush(TCX_MODULE_PTR module, int id)

tcxRecvFlush will clear out the message queue based on the values of module and id. If module is NULL and id is 0, then the entire queue is removed. Otherwise if module is set then only those messages from the module are removed. Similarly if id is the internal id value of a particular message then only that message will be removed. Used in combination a specific message sent from a specific module can be removed.

7 Message Handlers

7.1 Defining Message Handlers

TCX provides a method for creating a mapping between registered messages and a default procedure to be executed when a module receives that message. Handler registration provides this mapping. TCX supports two styles of handlers, one for the message layer and another for the data layer. The main difference in handler styles is the number and type of parameters passed to the handler on an incoming message. A data handler can be executed for a received message. Registration of data handlers is defined elsewhere. Only one handler can be registered per message definition.

```
typedef struct {
        char *msgName;
        char *hndName;
        void (*hndProc)();
        int hndControl;
        void *hndData;
} TCX_REG_HND_TYPE;
```

Each handler is identified by a message name is should be executed for, *msgName*, a string name for the handler for easy identification, *hndName* a pointer to the handler routine, *hndProc*, a value which controls how many times this handler can be called recursively, *hndControl* and finally an arbitrary pointer to data which will get passed to this handler each time it is called, *hndData*.

```
TCX_REG_HND_TYPE hndArray[] = {
        {"lineMsg", "lineMsgHnd", lineMsgHnd, TCX_RECV_ALL, NULL},
        {"pointMsg", "pointMsgHnd", pointMsgHnd, TCX_RECV_NONE, NULL},
        {"arrayMsg", "arrayMsgHnd", arrayMsgHnd, TCX_RECV_ONE, NULL}
};

RECV_ALL 1
RECV_NONE 2
RECV_ONCE 3
```

Handlers are triggered when a module receives a message. Modules receive messages by executing a TCX receive call. Handlers can also issue a TCX receive call which in turn may trigger the same or other handlers. The RECV_ALL parameter specifies that if a handler in turn executes a TCX receive call *All Handlers* are still valid for execution. The RECV_NONE parameter instructs the TCX receive call that *No Handlers* are to be executed until this handler is finished.The RECV_ONCE control value means the handler can not be executed recursively, all other registered handlers are still valid for execution during a TCX receive call.

Message handlers are called in the following manner:

```
handlerProc(TCX_REF_PTR ref, void *data, void *hndData)
```

The TCX_REF_PTR, *ref*, is a reference to the message being sent to this module. *data* is a pointer to the data sent to this module in a form specified by the message's format. *hndData* is a pointer to the data registered with the handler.

User data is created for a handler and should be freed unless needed. The call *tcxFreeByRef* can be used to free information created for handlers in the same fashion it was created. The parameter *ref* is freed when the handler procedure returns.

7.1.1 tcxRegisterHandlers

Register an array of handlers.

void tcxRegisterHandlers(TCX_REG_HND_TYPE *hndArray, int size)

Register a modules handlers.

```
TCX_REG_HND_TYPE hndArray[] = {
        {"lineMsg", "lineMsgHnd", lineMsgHnd, TCX_RECV_ALL, NULL},
        {"pointMsg", "pointMsgHnd", pointMsgHnd, TCX_RECV_NONE, NULL},
        {"arrayMsg", "arrayMsgHnd", arrayMsgHnd, TCX_RECV_ONE, NULL}
};
```

tcxRegisterHandlers(hndArray, sizeof(hndArray)/sizeof(TCX_REG_HND_TYPE))

7.1.2 tcxRegisterExitHnd

Register a routine to call on exit conditions.

void tcxRegisterExitHnd(void (*proc)())

By default TCX will call exit should a problem occur, *tcxRegisterExit* attempts to be die graceful.

7.1.3 tcxRegisterConnectHnd

Register routines to be called when a module connects or disconnects - needs work!

tcxRegisterConnectHnd(char *name, void (*openHnd) (), void (*closeHnd) ())

When a module named by *name*, connects call *openHnd*, when it disconnects call *closeHnd*. The handlers *openHnd* and *closeHnd* take one parameter, *char *name,* which is passed the name of the module connecting or disconnecting.

Because of auto-reconnect, connections and disconnections may occur during a *tcxInitialize* call, this routine may be called before *tcxInitialize*. *tcxRegisterConnectHnd* is the only routine which can be safely called before *tcxInitialize*.

Another complication is the names of modules may not be known before they connect. A more general solution is being discussed. In most cases a message sent by a module on start up can

accomplish all that a special connect handler could. tcxRegisterConnectHnd can be called at any-time to update a module's connection handlers.

7.1.4 tcxRegisterDHnd

Register a data handler.

void tcxRegisterDHnd(int id, TCX_FMT_PTR fmt, void (*proc)(), int control, int mask, void *data)

Avoid.

8 Data Layer

Much of the work of TCX is accomplished through the Data Layer. The major routine of importance is tcxRecvData, which probably does too much. This part of TCX is subject to the most change.

8.1 tcxSendData

Send information to a module.

void tcxSendData(TCX_MODULE_PTR module, int id, int ins, void *data, int len, TCX_FMT_PTR fmt)

Avoid.

8.2 tcxReplyData

&&&&

void tcxReplyData(TCX_MODULE_PTR module, int id, int ins, void *reply)

Avoid.

8.3 tcxRecvData

Receive information, return information, run handlers, establish connections,etc.

int tcxRecvData(TCX_MODULE_PTR *module, int *id, int *ins, void *data, TCX_FMT_PTR fmt, int allowHnd, void *timeout)

tcxRecvData is the most general form for receiving information. Most applications can get by making use of *tcxRecvMsg*. One particular call to *tcxRecvData* is worth mentioning is the ability to execute handlers without looking for a particular message. In general however this routine will probably be overhauled.

8.3.1 Timeout or Execute a handler

tcxRecvData will return after it has found the message of interest or if a handler has been executed. After each pass after gathering all messages pending, tcxRecvData will attempt to execute applicable handlers, if one or more handlers are run then tcxRecvData will return if it is not looking for a particular message. The return value in this case is -2. If a timeout is used in conjunction with this then an attempt is made to see if a message with a handler is pending, in this case -2 is returned otherwise a -1 timeout return value is returned.

The call tcxRecvLoop loops on calls to tcxRecvData. The actual call would follow this form:

tcxRecvData(NULL, NULL, NULL, NULL, NULL, ALL_HND, &timeout)

8.3.2 Selecting a particular Message or Module

If a NULL value is given for either module, or id then any information or id is acceptable.

tcxRecvData(NULL, NULL, NULL, &data, NULL, ALL_HND,NULL)

If module is equal to a module then that module must be the sender of the data received.

module = moduleAReference;

tcxRecvData(&module, NULL, NULL,&data, NULL, ALL_HND,NULL);

Similarly for the id value.

id = 17;

tcxRecvData(NULL, &id, NULL,&data, NULL, ALL_HND,NULL)

If the value of module is NULL or id is 0, then the value of the incoming message module or id is returned instead of being matched.

moduleRef = NULL;

idValue = 0;

tcxRecvData(&moduleRef, &idValue, NULL,&data, NULL, ALL_HND,NULL)

The above example would accept the next data item from some module whose module reference would be stored in moduleRef and whose message id would be stored in idValue. moduleRef can be used in a future call to tcxSendData without needing to connect to that module via tcxConnect-Module.

8.3.3 Controlling Storage Allocation

Whether or not space is allocated for data is controlled by the id value. If the id is a specific value in tcxRecvData then the parameter for data should point to an already allocated area of memory. This is done because if a specific id value is being searched for then it is assumed the caller knows the data storage required. If the value for id is 0 or the parameter for id is NULL, then it is assumed that the storage required for the data is unknown and should be created. In this case a pointer to the created data is returned.

8.3.4 Data Decoding

How the data is decoded is controlled by the value of the fmt parameter. If fmt is NULL then the information is byte copied into data, or a pointer to a buffer of the length sent is returned depending on the value of the parameter id. If fmt is not NULL then the fmt is used to decode the data.

When formatters are used only the top level structure needs to be allocated for the pre-allocated storage option. Formatters create dynamic storage for items within structures that need it.

If the value of the data parameter is NULL then no decoding of data is done but the amount of data sent is returned as the return value of the call.

If the value of fmt is the special value *tcxCreateFMT*, then data will be a pointer to the data read and the amount of data read will be returned by the call to *tcxRecvData*. If data was sent by a formatter, it can then be decoded by *tcxDecodeData*. If *tcxCreateFMT* is used then storage will be allocated to hold incoming data and data will be set equal to that storage.

8.3.5 Controlling Polling

tcxRecvData will block until the specified data is received and returns the number of bytes of data received (or the number of bytes of the encoded form if formatters are used). If the time parameter is not NULL but a valid timeval struct, this specifies a timeout value. If the timeout value is reached without finding the requested data, tcxRecvData returns -1.

8.4 tcxCreateFMT

Special flag formatter to control tcxRecvData decoding.

TCX_FMT_PTR tcxCreateFMT;

8.5 tcxDecodeData

Formatter internal decode routine.

void tcxDecodeData(TCX_FMT_PTR fmt, void *buf, void *data)

&&&&

8.6 tcxParseFormat

Turn a string formatter into its internal representation.

TCX_FMT_PTR tcxParseFormat(char *format)

&&&&

8.7 tcxFreeData

Clean up data using a formatter.

void tcxFreeData(TCX_FMT_PTR fmt, void *data)

&&&&

9 Error Recovery

9.1 Module Disconnections

Receive generally detects errors before sends

9.2 General Exit Handling

10 TCX_REF_PTR

10.1 tcxRefModule

Return the name of the sending module given a handler reference.

TCX_MODULE_PTR tcxRefModule(TCX_REF_PTR ref)

10.2 tcxRefId

Return the Id value of the message given a handler reference.

int tcxRefId(TCX_REF_PTR ref)

10.3 tcxRefIns

Return the message instance value given a handler reference

int tcxRefIns(TCX_REF_PTR ref)

10.4 tcxRefCopy

Copy a reference for future use. Handler References are cleaned up after the routine returns.

TCX_REF_PTR tcxRefCopy(TCX_REF_PTR ref)

10.5 tcxRefFree

Free a reference - Should only be used on those references created by tcxRefCopy

void tcxRefFree(TCX_REF_PTR ref)